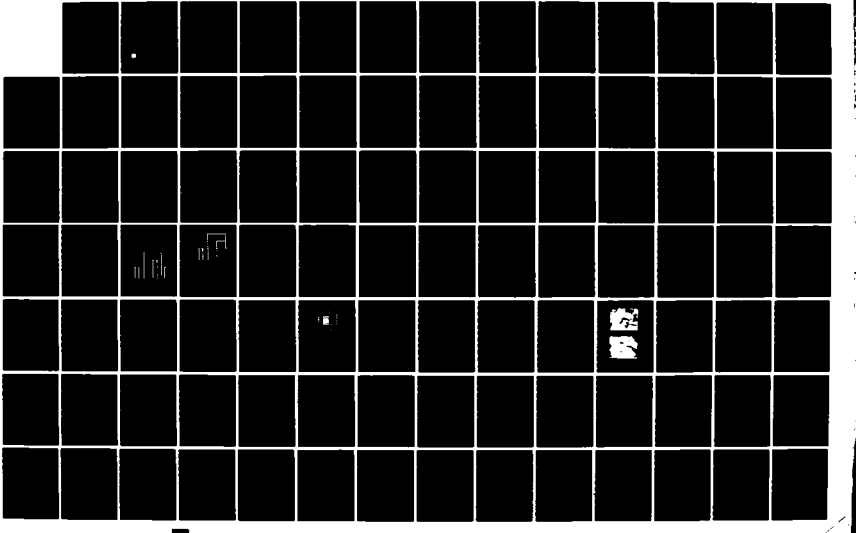
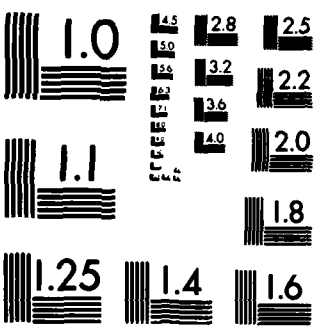
AD-A146 526 SPECIALIZED SILICON COMPILERS FOR LANGUAGE RECOGNITION (U) CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE M J FOSTER JUL 84 CMU-CS-84-143 1/2

UNCLASSIFIED F33615-81-K-1539 F/G 9/2 NL





COPY RESOLUTION TEST CHART

CMU-CS-84-143

13

Specialized Silicon Compilers for Language Recognition

July 1984

Michael J. Foster

DEPARTMENT
of
COMPUTER SCIENCE

Carnegie-Mellon University

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER CMU-CS-84-143	2. GOVT ACCESSION NO. AD-A146 526
	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) **SPECIALIZED SILICON COMPILERS FOR LANGUAGE RECOGNITION**	5. TYPE OF REPORT & PERIOD COVERED Interim
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) **Michael J. FOster**	8. CONTRACT OR GRANT NUMBER(s) **F3361581K1539, NR048659** **N0001476C0370,NR044422** **N0001480C0236**
9. PERFORMING ORGANIZATION NAME AND ADDRESS Carnegie-Mellon University Computer Science Department Pittsburgh, PA 15213	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research Arlington, VA 22217	12. REPORT DATE **July 1984** 13. NUMBER OF PAGES **113**
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in block 20, if different from Report)

Approved for public release; distribution unlimited

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

CMU-CS-84-143

Specialized Silicon Compilers for Language Recognition

July 1984

Michael J. Foster

Computer Science Department
Carnegie-Mellon University

This research was supported in part by the Defense Advanced Research Projects Agency (DoD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory under Contract F33615-81-K-1539, in part by the Office of Naval Research under Contracts N00014-76-C-0370, NR 044-422 and N00014-80-C-0236, NR 048-659, in part by the National Science Foundation under an NSF Graduate Fellowship, and in part by the Fannie and John Hertz Foundation.

Abstract

This thesis advocates the use of specialized silicon compilers in the design of high-performance custom VLSI circuits. A specialized silicon compiler is a design tool that accepts a behavioral specification for a circuit and produces the layout for a small, fast VLSI chip. Each specialized silicon compiler produces chips for only a small task domain. Because the task domain is restricted, a specialized silicon compiler can use application-dependent techniques for circuit design and layout, thus ensuring the efficiency of the chips that it produces.

The major portion of this thesis describes a specialized silicon compiler that generates recognizers for regular languages. Given a regular expression describing a language to be recognized, the compiler automatically produces the layout for a high-speed recognizer. Besides being a prototype of a useful tool for designing recognizers, this compiler serves as a model for compilers specialized to other areas. It is used to illustrate techniques for construction and verification of specialized silicon compilers.

The thesis also describes a specialized programmable layout for language recognizers. A specialized programmable layout is a chip that is designed as a target for a particular specialized silicon compiler. Parts of the circuit that are the same for all problems in the task domain are laid out in advance, while parts of the circuit that vary from one problem to another are left to be programmed. The layout described in this thesis has cells for primitive recognition operations laid out in advance and is programmed for a particular regular language by interconnecting these cells. This layout was implemented in NMOS and is programmed after fabrication by cutting metal lines using a laser.

A-1

This thesis is submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.

Acknowledgements

The aid and advice of many competent and generous people made this thesis possible. I thank them all. This work could not have been accomplished without the assistance of my advisor, H. T. Kung. By example and advice, he showed me how to do effective research. The remaining members of my committee, Allan Anderson, Bob Sproull, and Guy Steele, made important contributions to the content and clarity of this document. Allan and his colleagues Glenn Chapman and Jack Raffel helped me design, program and test the laser-programmable E.T. chip. Tom Nuhfer helped me use a scanning electron microscope to diagnose the E.T. chip. Roland Backhouse and David Muller saved me from embarrassment by finding errors in early versions of this work. Cynthia Hibbard helped revise the final draft of this document, greatly improving its clarity. Ed Frank grappled with our photocomposer so that I could include detailed checkplots. I am grateful to Jon Bentley, Peter Dew, Merrick Furst, Charles Leiserson, Amar Mukhopadhyay and Jeffrey Ullman for insightful comments on some of the issues in language recognition. Finally, to my friends in Pittsburgh who helped me through the rough spots, my heartfelt gratitude.

Table of Contents

Chapter 1
Introduction

Custom VLSI holds great promise for solving computationally demanding problems in a cost-effective manner. Modern MOS processes allow 10^5 to 10^6 devices to be fabricated on a single chip, and this number is expected to increase by another order of magnitude during the next decade. For some applications, a few custom chips may be as effective as a large supercomputer, at a far lower cost.

Before this promise can be realized, however, the main impediment to the use of custom VLSI must be addressed. This is the high cost of chip design, which is due to the complexity of the chip design process. To build an efficient chip, a designer must think about a wide range of disciplines, from the underlying algorithm for the chip to its final layout. Design errors are frequent and the turnaround time for corrections may be as long as several months. If custom chips are to come into widespread use, methods must be found to manage this complexity and to detect errors at an early stage. Ideally, designing and debugging an efficient custom chip should be no more difficult or costly than constructing software to solve the same problem.

To reduce the complexity of custom VLSI design, a design tool is needed that automatically lays out an efficient custom chip from its behavioral specification. Such tools, often called *silicon compilers*, already exist, but while they fulfill the requirement of performing automatic layout from a behavioral specification they fail to produce chips that are efficient. Using these tools, the behavioral specification for a chip can be written in a high-level programming language, and can be checked and modified before the chip is laid out. Automation of the layout process ensures that the chip and the program have the same behavior. This eases the design of custom chips in two ways.

- Circuit design, layout, and similar low-level design tasks are eliminated.
- The number of design iterations is reduced, since finished chips are likely to meet their behavioral specifications.

Although this automation of the design task is desirable, efficiency of the final chips is essential. To

be useful, silicon compilers must produce chips that are nearly as small and fast as those that can be designed by hand. If silicon compilers that produce efficient chips from behavioral specifications can be built, custom VLSI will become more useful.

1.1. Specialized Silicon Compilers

To address the problem of producing efficient custom chips, this thesis proposes the use of *specialized silicon compilers*. A specialized silicon compiler produces chips for only a small domain of tasks. Within this domain, it produces efficient chips automatically, from their behavioral descriptions. By forsaking generality, specialized silicon compilers gain efficiency. This thesis contributes to the design of custom VLSI by identifying a general structure for specialized silicon compilers and presenting a particular compiler as an example.

The overall structure of a specialized silicon compiler is independent of its task domain. It contains layouts of some primitive components, or *cells*, along with methods for using the cells for specific problems. A specialized silicon compiler for VLSI has three parts:

- An *application area*, or set of problems for which the compiler is intended;
- A set of layouts for primitive *cells*;
- *Rules* for specifying problems in the application area, and for combining the cells to solve a specified problem.

Although the individual cells, rules, or application areas may differ from one compiler to another, every specialized silicon compiler has this three-part structure.

This three-part structure gives specialized silicon compilers their power. Chips produced by these compilers can be efficient, because the primitive cells and methods for interconnecting them can be carefully designed. At the same time the design process can be simple, because the rules automate the translation from problem specification to chip layout. With a specialized silicon compiler, a chip designer can use small, fast circuits with minimal design effort. The combination of well-designed cells and application-specific rules aids the rapid design of efficient chips.

Another benefit of the three part structure of rules, cells, and application areas is ease of construction and maintenance of specialized silicon compilers. This structure partitions the compiler into one part that is in the domain of LSI designers (the cells) and a second part that is in the domain of application experts (the rules). This partitioning permits experts in many fields to participate in constructing VLSI design tools, so that each part of a compiler can be built by the most qualified

people. The resulting division of labor simplifies the initial construction of a specialized silicon compiler. Furthermore, modification of a compiler may require altering only a few rules or cells. Changing fabrication technologies, for example, would require changing only the primitive cells of a compiler. The set of rules could remain unchanged. Because of these features, specialized silicon compilers are easy to create and modify.

Still another advantage of the three-part structure is the verifiability of specialized silicon compilers. The correctness of the circuits produced by a compiler can be verified by formal methods. Each primitive cell can be checked independently of the others. When all cells are correct, the rules that translate problems in the application area to layouts can be checked using the method of Chapter 5 to make sure that the layouts exhibit the correct behavior. Specialized silicon compilers thus help to ensure correct chips.

Several alternatives to specialized silicon compilers have been explored. These alternatives fall into two categories: general silicon compilers, and automatic layout systems. General silicon compilers produce layouts from behavioral specifications, while automatic layout systems produce layouts from logic diagrams or similar structural specifications. The following discussion will show that neither of these types of design tools addresses the complete problem of automatically translating behavioral specifications into efficient custom chips.

A few general silicon compilers have been built [26, 46, 70, 72] which accept a behavioral description of a circuit, typically in a high-level programming language, and produce a chip meeting that description. Despite the benefits of automation that they provide, silicon compilers cannot be used in many applications because of the inefficiencies of the chips they produce [79]. Specialized algorithm, circuit, and layout techniques must be used in some application areas; general silicon compilers do not include enough knowledge to use these techniques in every case. Silicon compilers that are not specialized fail to produce efficient custom chips.

Automated layout systems have been constructed that take a wide range of structural descriptions as input. Chip assemblers [41] produce layouts given a set of custom-designed leaf cells, together with a global floorplan. Placement and routing systems [6, 24, 45, 64] combine predefined library cells that act as gates and registers into chip layouts, given a gate-level description of the chip. Compaction programs [33, 80] and matrix layout systems [54, 78] help in producing layouts from circuit diagrams. All of these types of automated layout systems can produce small, fast chips (although not in all applications). However, they all require the designer to make some structural decisions about the chip to be designed. They do not translate behavior to layout.

A specialized silicon compiler, however, combines the advantages of both general silicon compilers and automatic layout systems. Design times are short, because the behavior of the chip is the only specification. Efficiency of the final chip can still be high, though, because specialized knowledge of the problem domain can be included in the compiler. By exploiting the power of specialization, these compilers can make custom VLSI feasible for many new tasks. Construction of specialized silicon compilers for common application areas will shorten chip design time, and fulfill the promise of VLSI.

1.2. Previous Work with Specialized Silicon Compilers and VLSI Building Blocks

Although a structure for specialized silicon compilers had not been identified before this thesis, a few existing VLSI design tools can be seen in retrospect to be specialized silicon compilers. These tools include specialized cells together with rules for connecting them together for specific problems in a task domain. In addition to these compilers, several *building block* systems have been proposed and built. Like a specialized silicon compiler, a building block system includes a set of cells that are tailored for a specific application area. Unlike a specialized silicon compiler, however, a building block system doesn't include explicit rules for interconnecting the cells. The user of the building block system must know how to compose the cells for an application; a structural description of the circuit to be built is needed. This section surveys some of the specialized silicon compilers and building blocks that have been proposed for custom VLSI design.

Compilers similar in intent to the one described in this thesis have been built by Ullman and his colleagues [42], and by Philipson at the University of Lund [65]. These compilers accept regular expressions as input and produce PLA-based layouts of recognizers for those expressions. Although these specialized silicon compilers have the same application area as the one described in this thesis, they have different rules and cells. The cells for these systems are parts of PLA's and registers, while the rules direct the construction of a finite-state machine for the regular language and the realization of that machine using the cells. Chapter 4 describes in greater detail the differences between the compiler presented in this thesis and these finite-state machine compilers.

Another silicon compiler, which can be thought of as a specialized compiler for microprocessor-like circuits, is MacPitts [70, 72]. It translates a program written in a dialect of LISP into a chip that is based on a two-part "target architecture." The target architecture consists of a data-path implemented with bit-sliced cells and a control section built with array logic. This target architecture is appropriate for many digital systems, though it must be extended for some application areas [31].

In a sense, MacPitts is a specialized silicon compiler for programs that can be implemented efficiently on that target architecture.

Building blocks have a longer history than specialized silicon compilers and have been successfully used in several applications. One area in which building blocks are common is digital signal processing. Lyon [56] has presented an architectural framework for the design of bit-serial signal processors. He gives examples of useful primitive components (such as multipliers, adders, and second-order filters) and describes a simple technique for specifying their interfaces. Denyer and Myers [23] show how bit-serial arithmetic can be pipelined to make digital filters nearly as fast as those implemented with parallel arithmetic. Building upon this work, Bergmann [7] describes a compiler that translates signal flow graphs into layouts. This is still a structural description to some extent, however, since a signal processing operation may have several flowgraphs [19].

A second application area for building blocks is the use of regular arrays such as PLA's, Weinberger arrays, and gate arrays to implement combinational logic [68]. The primitive cells in these logic arrays represent parts of the logic equations such as a variable included in a product term. The cells are laid out according to a truth table derived from the equations. Array logic systems are classified as building blocks rather than as specialized silicon compilers because the logic equations that serve as input are often conceptually distant from the desired behavior. Rules for generating logic equations from a behavioral specification are not included in these systems.

A third set of building blocks is used in data paths for microprocessors [2]. Primitive components, such as bit-sliced ALU's and register files, are placed on a fixed-format layout. Input to this program is a register-transfer language that describes the structure of the data-path. Though this is a structural description, it is at a high level of abstraction. Extensions of current research [35, 46] may show how to translate a behavioral description of a data path into an efficient structure. This research could generate the set of rules needed to convert this building block system into a specialized silicon compiler.

1.3. This Presentation

The major portion of this thesis describes a specialized silicon compiler. This compiler generates recognizers for regular languages, using a small set of primitive cells and syntax-directed rules for interconnecting them. Besides being a prototype for a useful tool, this example serves as a model for compilers specialized to other application areas.

Chapter 2 describes a circuit compiler for systolic recognizers. Gate-level designs of the primitive cells are presented, along with a small set of rules for interconnecting them. Some extensions to language recognizers are presented that could increase the area of applicability of these circuits while not unduly increasing their complexity.

Chapter 3 discusses the layout of the language recognizers from Chapter 2. Selection of one of the layout methods in Chapter 3 converts the circuit compiler into a specialized silicon compiler. The chapter concentrates on specialized programmable layouts for language recognizers. A specialized programmable layout is a chip that is designed to work with a particular specialized silicon compiler. The primitive cells are laid out in advance and the interconnection rules are used to program the layout. Programmable layouts can amplify the advantages of specialized silicon compilers. The chapter ends with a description of a laser-programmable layout implemented in NMOS.

The compiler described in Chapters 2 and 3 produces recognizers that use a single recognition algorithm. A more complete compiler might choose between the many algorithms available for recognizing regular languages. Chapter 4 surveys these algorithms, and suggests criteria by which a compiler might choose between them.

Chapter 5 introduces a method of verifying the correctness of specialized silicon compilers and gives several examples of its use. If the construction rules of the compiler are expressed using an attributed context-free grammar, the functional correctness of circuits constructed by the compiler can be verified mechanically. The usefulness of this technique suggests that the methods of this thesis should be widely applied, so that specialized silicon compilers will be correct and comprehensible.

Chapter 6 summarizes the results of the thesis and suggests directions for further research.

Chapter 2
A Specialized Circuit Compiler for Language Recognizers

This chapter describes a specialized silicon compiler that constructs systolic recognizers for regular languages. The compiler consists of several primitive cells, together with a procedure for hooking them together to create a recognizer for a given regular expression. Several extensions to the compiler are described that allow smaller recognizers for some languages or additional circuit functions.

Although several characterizations of regular languages are possible [37, 67], this thesis defines a regular language as a set of strings (possibly including the empty string ϵ) from a finite alphabet Σ that can be specified by a regular expression over Σ. A regular expression may represent the empty set (φ) or any set of strings that can be built up by concatenation, union and repetition from the empty string ϵ and the single characters of Σ. A regular expression over Σ may include some characters that are not in Σ, such as operators and parentheses. Assuming that the characters in the set $\{\varphi\ (\ )\ *\ +\}$ are not in Σ, the syntactically correct regular expressions over Σ can be defined inductively as follows.

- φ is a regular expression over Σ.
- If $a \in \Sigma$ then a is a regular expression over Σ.
- If α and β are regular expressions over Σ, then so are $\alpha\beta$, $(\alpha + \beta)$, and $(\alpha)^*$.

The meaning of a regular expression can be defined inductively based on the form of the expression. The set of strings $L(\rho)$ represented by a regular expression ρ is:

- The empty set if ρ is φ.
- $\{a\}$ if ρ is a.
- $L(\alpha) \cup L(\beta)$ if ρ is $(\alpha + \beta)$.

- $\{\sigma_1\sigma_2$, where $\sigma_1 \in L(\alpha)$ and $\sigma_2 \in L(\beta)\}$ if ρ is $\alpha\beta$.
- $\{\varepsilon\} \cup \{\sigma_1\sigma_2 \ldots \sigma_n$, where n is any positive integer and $\sigma_i \in L(\alpha)\}$ if ρ is $(\alpha)^*$. In this thesis, λ will be used as an abbreviation for φ^*. Thus, $L(\lambda) = \{\varepsilon\}$.

This thesis uses a regular expression to denote the set of strings it represents; for example, $abc \in (ab + c)^*$.

Originally introduced to describe nerve nets [43], regular languages have seen wide application in computer science. They have been used to specify lexical analyzers for programming languages [53], controllers for sequential machines [28, 76], filters for on-the-fly database search [34], patterns in image processing [40], and communication protocols [36]. Regular expressions that have been augmented in various ways have been used in speech recognition [55], process synchronization [3, 14], hardware testing [9], and program debugging [12]. Circuits that can be specified by regular expressions thus form a large and interesting problem domain, and a specialized silicon compiler for this domain should prove useful.

In describing circuits, it is essential to specify their input-output behavior. The pattern matching and recognition algorithms described in this chapter will all use the same behavior. All circuits will operate in discrete time steps, called *beats*. The regular expression describing the pattern will be specified in advance, then a string will be input to the recognizer one character at a time. No more than one character is input on a single beat. The recognizer is expected to output a bit after each character, telling whether it is the last character of a recognized substring. Both the string and the result stream are thus viewed as time series, rather than as characters printed on a page, which could be seen all at once.

With this input-output behavior, in which the string is presented as a time series, any recognition algorithm requires $\Omega(n)$ time to find all matches in a string of length n, even with unbounded parallelism. That much time is required just to read the string. In contrast, if all characters of the string were available at the start, the matching substrings could be found in time $O(\log n)$ by parallel composition in the syntactic monoid [20, 21]. The circuits described in this chapter overlap input-output with computation, so that they operate in $O(n)$ time.

2.1. A Compiler For Systolic Recognizer Circuits

The compiler described in this chapter produces a systolic recognizer for regular languages by connecting together primitive cells. The advantages of systolic algorithms in VLSI have been discussed extensively [29, 51, 50], and include high throughput, quick response time, ease of layout, and extensibility. These advantages arise from the regular layout and local, broadcast-free communication of systolic algorithms.

A comparison of systolic and non-systolic circuits for matching simple text patterns will illustrate these advantages and motivate the construction of a systolic recognizer for regular languages. Figure 2-1 shows a non-systolic pattern matcher [58]. Characters from the string are input into a shift register, each stage of which is associated with a stored pattern character and a comparator, as shown in Figure 2-2. (In Figure 2-2 and throughout this thesis, shift register stages or one-beat delays are shown as boxes containing the character "Δ".) On each beat, a character is input and the shift register shifts left to receive it, after which all comparisons take place in parallel. The results of the comparisons are combined using an AND gate to form the result of comparing the pattern to one text substring.

A disadvantage of this non-systolic pattern matcher is the large fan-in needed for combining results. The AND gate in Figure 2-1 requires as many inputs as there are stages in the shift register. Unbounded fan-in of this type, or the global broadcast that appears in other non-systolic pattern matchers [60], can degrade performance and lead to routing problems in very large circuits.

Figure 2-3 shows a systolic pattern matcher for the same problem. The shift register for characters still shifts leftward one stage on each beat, but characters are separated by an extra stage, so that cells alternate between activity and idleness. The multi-input AND gate is replaced by a shift register for partial results, which shifts rightward on each beat. As shown in Figure 2-4, each cell that is active on a beat compares the text character with its stored pattern character, combines the result of that match with its result input, and shifts the updated result out to the right.

Each cell in Figure 2-3 communicates only with its nearest neighbors, and no unbounded fan-in or broadcast is required. The advantages of the systolic algorithm are evident.

- Layout is simplified, since no global signal paths are needed.
- Speed is easier to achieve, since there are no time-consuming broadcasts or large fan-ins.
- The circuit is easy to extend, since only a small number of connections must be made to a new cell.

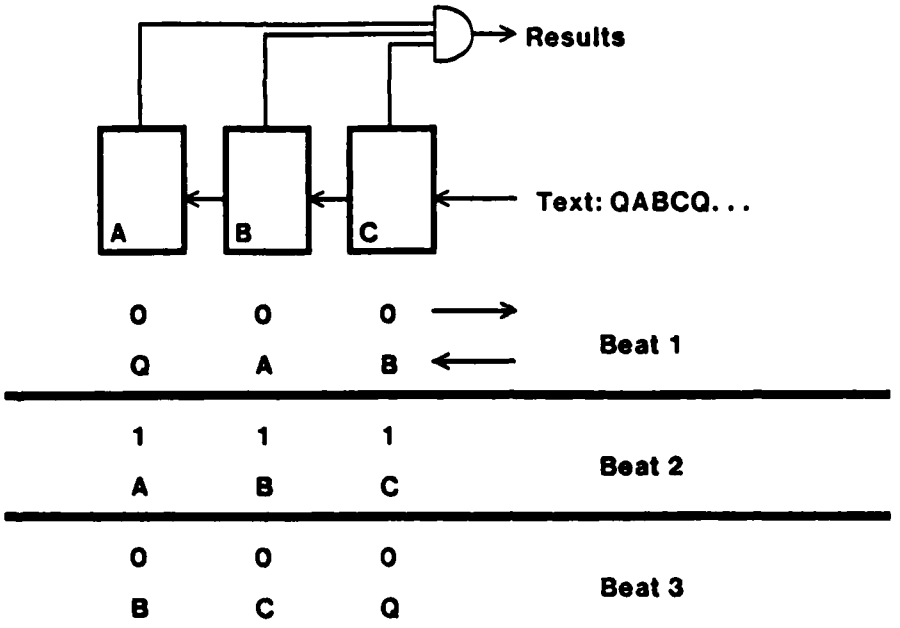


Figure 2-1: A non-systolic pattern matcher for ABC

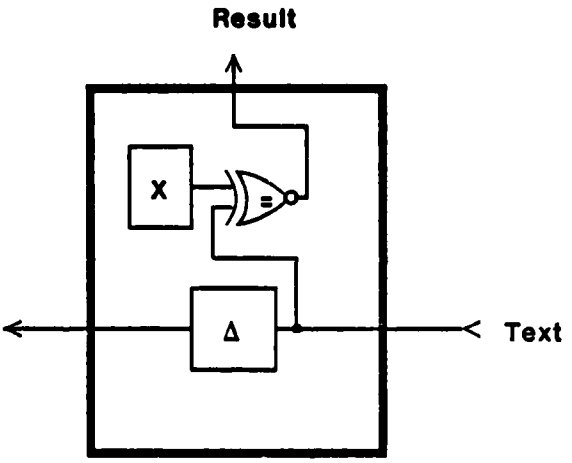


Figure 2-2: Cell for non-systolic pattern matcher

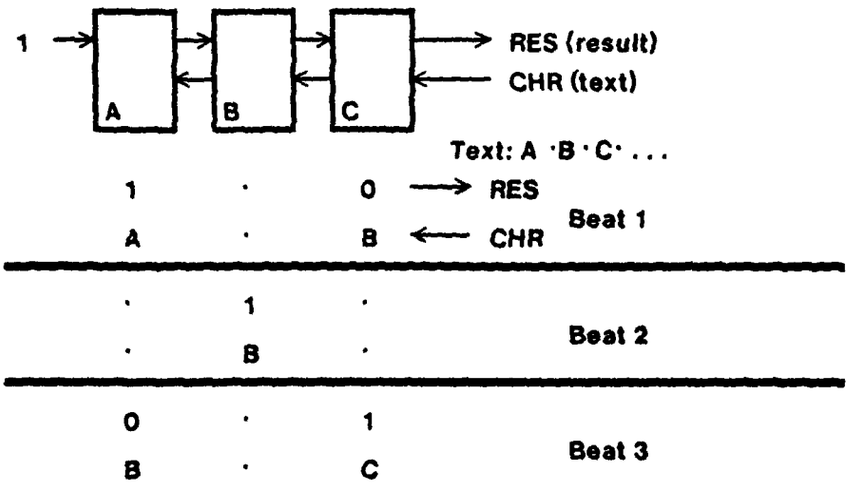


Figure 2-3: Systolic pattern matcher for ABC

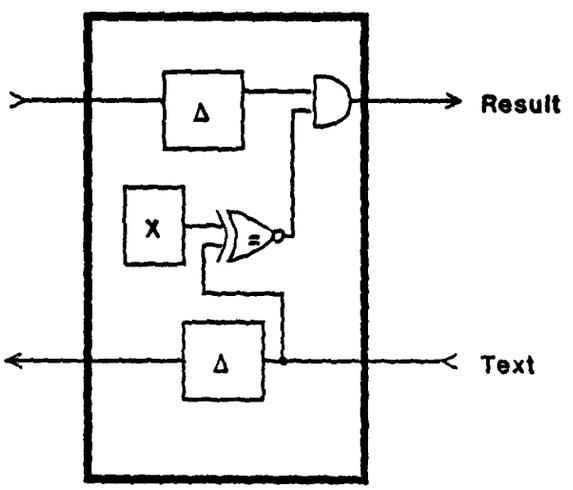


Figure 2-4: Cell for systolic pattern matcher

The systolic algorithm appears to have several disadvantages; the additional delay element in each cell might complicate the hardware and the extra separation between characters might slow down the data rate. Furthermore, half of the cells are idle on each beat. These disadvantages can be eliminated during implementation. If dynamic storage is used in NMOS, no additional hardware is needed for the shift registers [29].

- The additional inverter needed in the systolic algorithm replaces the bus driver needed in the non-systolic algorithm.
- The alternation of active and idle cells corresponds perfectly to the alternation of active and idle inverters in the shift registers.
- The equality gate can be shared between adjacent cells.

On balance, the advantages of the systolic algorithm seem to outweigh the disadvantages.

My recent work with H. T. Kung [30] shows how to extend the systolic pattern matcher of Figure 2-3 to recognize regular expressions, while maintaining the advantages of systolic algorithms. The first step is to add an enable signal, replacing the constant 1 input at the left of Figure 2-3. Figure 2-5 shows the character cell with an added enable signal, which is simply a one-stage shift register. (An abbreviating symbol for the character cell is also shown in the figure.) A pattern matcher is built from these cells by stringing them together and *terminating* the leftmost cell — connecting its enable output to its result input. On odd beats, values of ENB are input at the right of the pattern matcher and values of RES are output. On even beats, characters are input on the CHR data path. The resulting pattern matcher outputs 1 on RES if and only if the correct string is input on CHR, immediately preceded by a 1 on ENB.

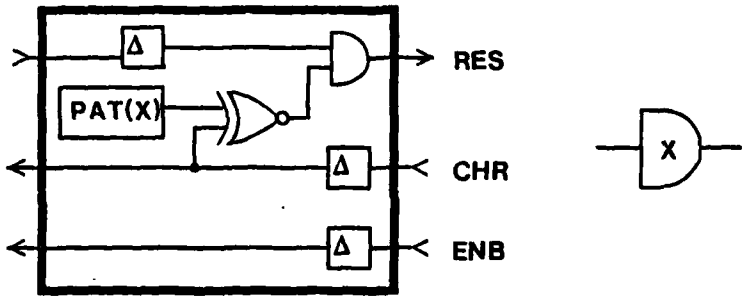


Figure 2-5: Character comparator cell with enable

A behavioral description of pattern matchers built from the cell of Figure 2-5 will make the interactions among the signals RES, ENB and CHR clearer. A pattern matcher for a pattern of length n is a circuit with two inputs ENB and CHR, and one output RES. On odd beats, ENB and RES are valid,

while CHR is valid on even beats. The output at time t, RES_t, is 1 if and only if ENB_{t-2n} is 1, and the string $\langle CHR_{t-2n+1}\ CHR_{t-2(n-1)+1}\ \cdots\ CHR_{t-1}\rangle$ matches the pattern. Table 2-1 traces the operation of a pattern matcher for several beats.

Beat	ENB	CHR	RES	Comment
1	1		0	
2		a		
3	1		0	
4		b		
5	0		0	
6		c		
7	0		1	A match, enabled at beat 1
8		a		
9	0		0	bca doesn't match the pattern
10		b		
11	0		0	Not enabled on beat 5, and no match
12		c		
13	0		0	Not enabled on beat 7

Table 2-1: Trace of a pattern matcher for abc

The behavioral description of pattern matchers can be extended from simple patterns to regular languages. The output RES_t of a recognizer for a regular expression ρ will be 1 if and only if there is some $n \geq 0$ such that ENB_{t-2n} is 1, and $\langle CHR_{t-2n+1}\ CHR_{t-2(n-1)+1}\ \cdots\ CHR_{t-1}\rangle \in L(\rho)$. It remains to show how to construct such a systolic recognizer for any regular expression.

The basic idea behind the systolic recognizer circuits is the decomposition of a regular expression into a concatenation of simpler subexpressions. A recognizer is constructed for each subexpression and these recognizers are interconnected into a pipeline similar to that shown in Figure 2-3. A syntax-directed technique, based on the generating grammar for regular expressions, allows automatic construction of recognizers. This syntax-directed technique can be easily extended by adding more productions to the grammar, and its correctness can be verified using the techniques of Chapter 5. Using this technique, then, a correct, flexible specialized compiler can be built.

The syntax-directed procedure for constructing recognizers uses a grammar that generates regular expressions, associating a part of the construction procedure with each part of the grammar. Each terminal symbol in the grammar corresponds to a primitive cell, each non-terminal corresponds to a more complex combination of cells, and each production corresponds to a construction rule. A recognizer is built by parsing the regular expression and following the construction rules associated with the productions used during the parse. When a terminal symbol is reached during the parse, the corresponding primitive cell is added to the circuit.

The following grammar for regular expressions is used to construct systolic recognizers.

$R \rightarrow P \mid RP$

$P \rightarrow \varphi \mid \langle letter \rangle \mid (R + R) \mid (R)^*$

The terminal symbols of this grammar are φ (the null expression), the letters of Σ, and the symbols "+" and "*". The non-terminal symbols are R, corresponding to a regular expression, and P, corresponding to a *primitive regular expression* (an expression for which concatenation is not the top-level operator). The grammar generates a regular expression as a concatenation of subexpressions, none of which has concatenation as its top level operator.

A primitive cell is needed for each terminal symbol of the grammar. The cell for a letter has already been introduced, and is shown in Figure 2-5. The cell for φ simply outputs false on RES, as shown in Figure 2-6. The cell for "+", shown in Figure 2-7, enables the recognizers for its operands using RES from the left, and OR's the results from its operands to produce its own RES. Operands for the "+" cell are connected to the top and bottom of the cell shown in Figure 2-7.

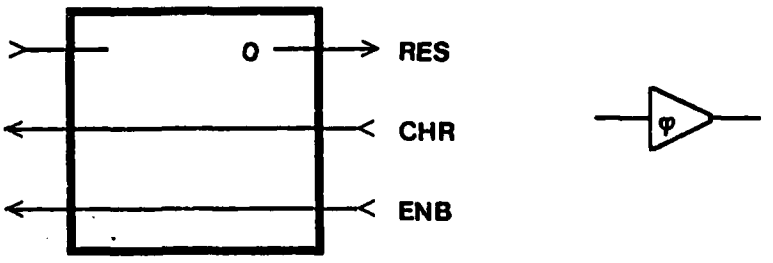


Figure 2-6: φ cell

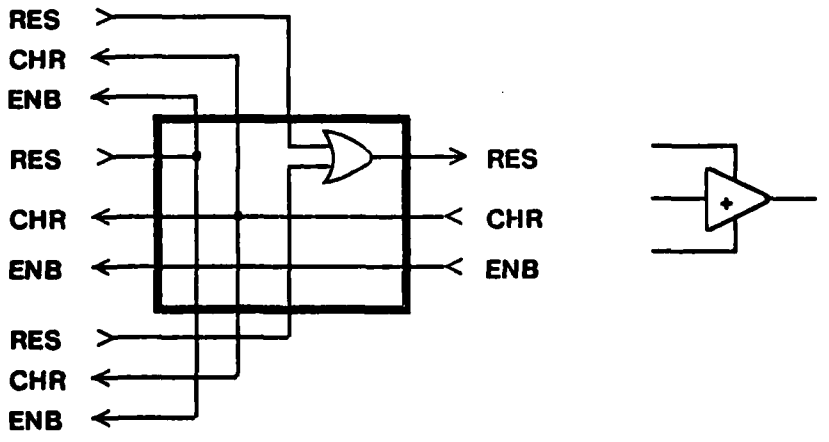


Figure 2-7: "+" operator cell

The cell for the Kleene * requires a non-standard gate, called a *clocked or* gate. This gate outputs the OR of its two inputs, except for a brief time between beats when it outputs 0. Using the two-phase non-overlapping clocking scheme often used in simple NMOS circuits [57], a clocked OR gate can be made with only a few transistors more than a combinational OR gate. For example, in the prototype chip described in Section 3.3, a beat consists of one φ_1 and one φ_2 phase. Shift registers similar to Figure 2-8 self-refresh on φ_1 and shift data on φ_2. Notice that the output of a one-beat delay remains stable throughout φ_2. For this clocking scheme, the circuit of Figure 2-9 acts as a clocked OR gate. Similar circuits can be constructed for other clocking schemes.

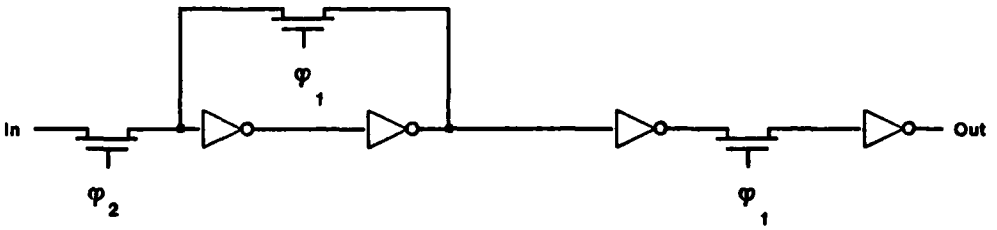


Figure 2-8: One beat delay (Δ) using two-phase clocking

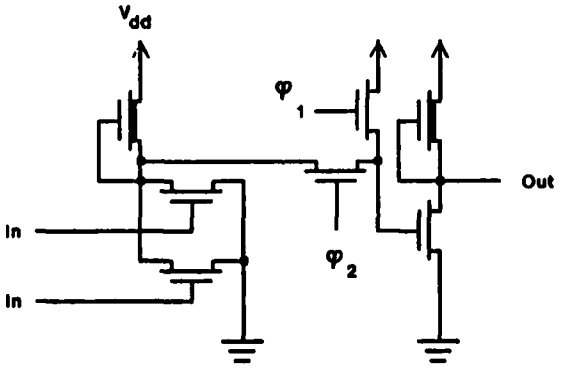


Figure 2-9: Clocked OR gate for two-phase clocking

Using the clocked OR gate, a cell for the Kleene * can be built as shown in Figure 2-10. The recognizer for the operand of the * is connected to the top of the cell. This circuit sets its RES_{out} to the OR of its RES_{in} and its operand's RES_{out}. Any time RES_{out} is true, it enables its operand to look for another instance. The clocked OR gate is used instead of a normal combinational OR gate to avoid latch-up problems between cross-coupled OR gates. If the operand of a Kleene * cell is another Kleene * cell, for example, the OR gates in the two cells feed back into each other as shown in Figure

2-11. In fact any expression of the form $(E)^*$, where $\epsilon \in E$, results in a cycle of OR gates [4]. This causes no problem when clocked OR gates are used.

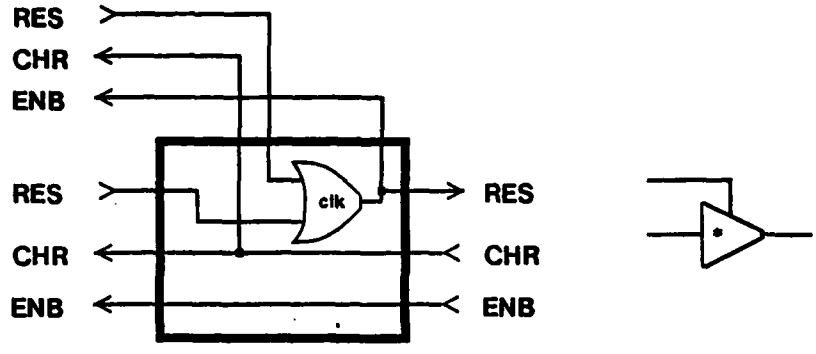


Figure 2-10: "*" cell

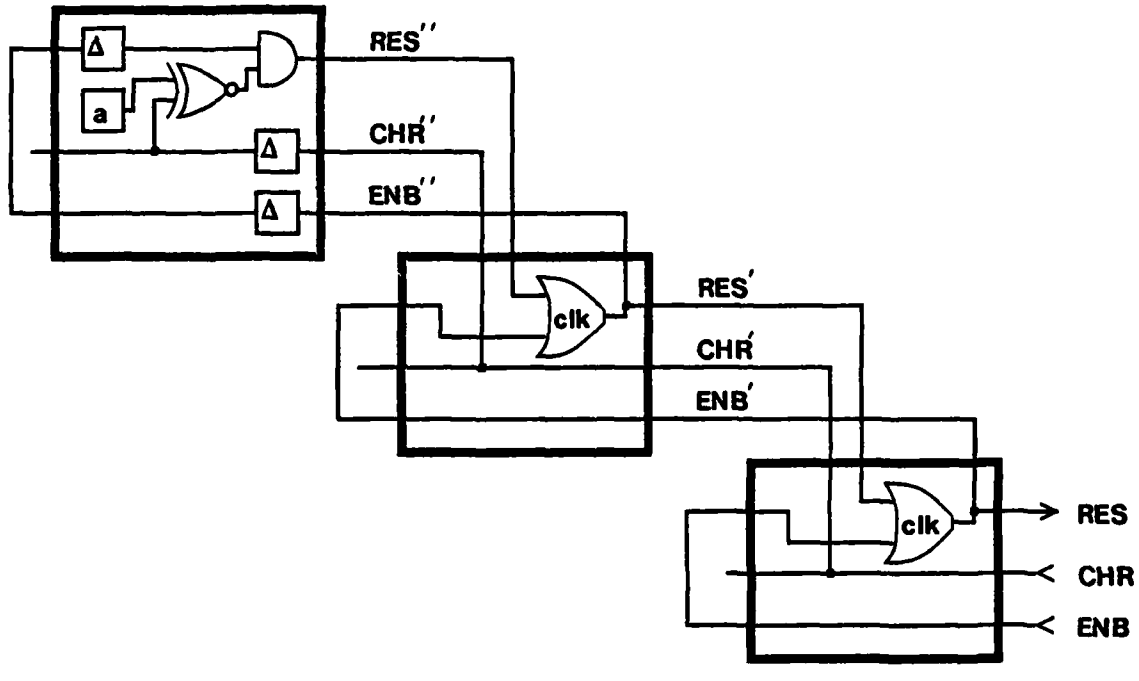


Figure 2-11: Recognizer for (a*)*

To see why the clocked OR gate eliminates the latch-up problem while still maintaining circuit correctness, consider the rightmost "*" cell in Figure 2-11. Suppose that the ENB input to this cell is 1 on beat 1, and 0 on all subsequent beats. Then on beat 1, the RES output is 1, which sets the RES'

output from the middle "*" cell to 1; the OR gates form a latch which stays at 1, even if ENB goes to 0 during the beat. Because these are clocked OR gates, however, their outputs are forced to 0 before the start of beat 2. Since ENB is 0 on all further beats, RES can be set to 1 only if RES' from the middle cell becomes 1 first. The transition of RES' to 1 must *precede* that of RES. For RES' to turn on, then, RES'' from the comparator cell must be 1. Hence, the only way that RES can be 1 on a beat is if RES'' from the comparator cell first becomes true on that beat. But in that case, a string of non-zero length must have been recognized. Conversely, any time a string of non-zero length is recognized, the RES output will be set to true. The clocked OR gate thus ensures correctness of the circuit by ensuring that a match of a non-empty string occurs between outputs of 1 on RES.

To connect the primitive cells together into recognizers, a set of construction rules is associated with the productions of the grammar. These rules tell which ports of each circuit are to be connected together, and which ports are to be terminated. All cells in Figures 2-5 through 2-10 have left and right ports. Some cells have upper and lower ports as well, for the connection of operands. The compound circuits corresponding to the nonterminals P and R may inherit left and right ports from their constituent cells. For example, any primitive recognizer (denoted by the non-terminal symbol P) has a left and right port while a recognizer (denoted by R) has only a right port. The six productions, with their semantic actions are:

$R \rightarrow P$	Terminate the left port of the circuit for P by connecting ENB_{out} to RES_{in}
$R \rightarrow RP$	Connect the left port of P to the right port of R.
$P \rightarrow \varphi$	Use a new φ cell as the circuit for P.
P → <letter>	Use a new comparator for P.
$P \rightarrow (R + R)$	Connect the right ports of the R's to the top and bottom ports of a new or-node.
$P \rightarrow (R)^*$	Connect the right port of R to the top port of a new star-node.

Figure 2-12 shows the syntax-directed construction of a recognizer for the expression (ab + (c)*). The expression is parsed top-down, and the semantic actions and cells described above are used.

One detail remains to complete the description of our systolic recognizers: they must be initialized. Before beginning operations, a RESET signal must be sent to all comparators. The RESET signal simply sets all shift register stages to 0. This ensures that no string not in *R* is recognized by *R*'s recognizer. If, for example, the RES shift register stage in the A cell of a recognizer for ABC contained 1 at the start of operations, the circuit would recognize the string BC

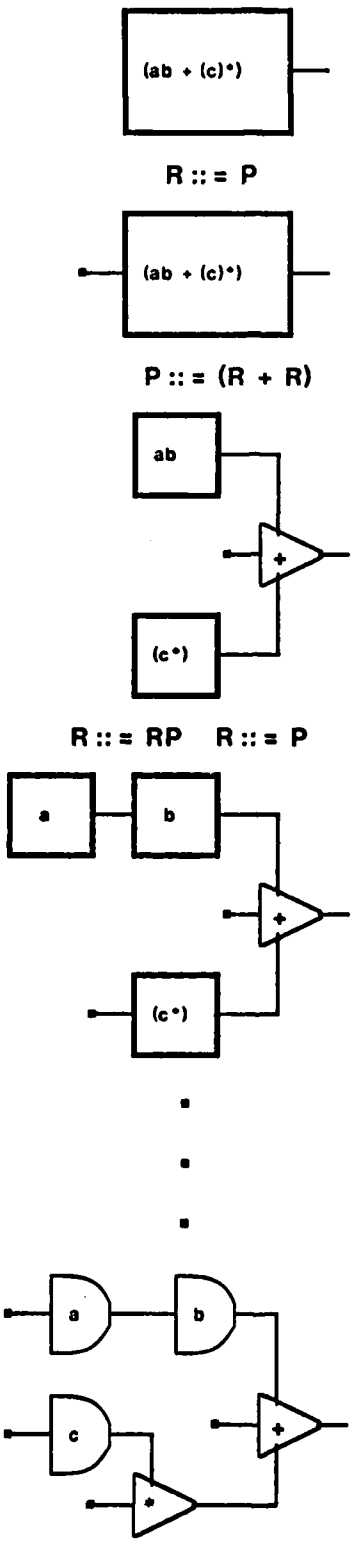
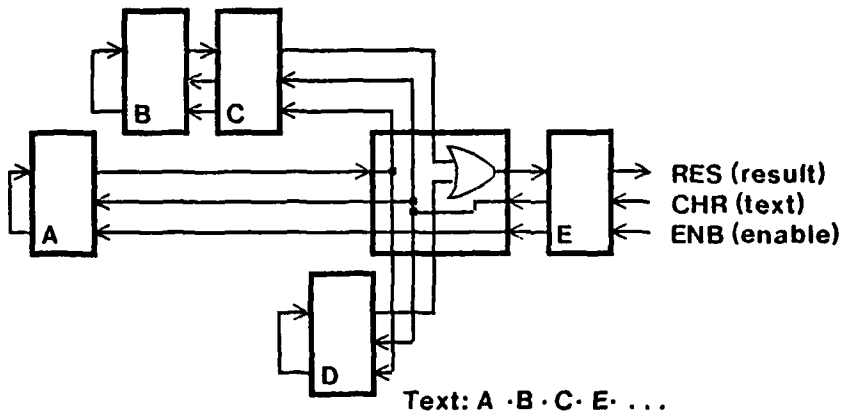


Figure 2-12: Construction of a recognizer for (ab + (c)*)



Text: A ·B · C· E· . . .

[1]	·	0	0	·	→ RES
A	·	A	A	·	← CHR
·	0	·	·	1	← ENB (= 1)
·	0	·	·	0	
·	A	·	·	B	Beat 2
1	·	[1]	1	·	
0	·	0	0	·	
B	·	B	B	·	Beat 3
·	[1]	·	·	1	
·	[1]	·	·	0	
·	B	·	·	C	Beat 4
1	·	0	0	·	
0	·	[1]	0	·	
C	·	C	C	·	Beat 5
·	0	·	·	1	
·	0	·	·	[1]	Result = 1
·	C	·	·	E	Beat 6
1	·	0	0	·	

Figure 2-13: Systolic recognizer for A(BC + D)E

Recognizers constructed using this syntax-directed procedure meet the behavioral description set out earlier in this section. If 1 is input on the ENB stream of an initialized recognizer, followed by a recognized string, then 1 will be output on the RES stream on the beat immediately following the last character of the string. Otherwise RES outputs 0. Figure 2-13 shows the operation of a systolic

recognizer over several beats, with the 1's in boxes tracking a successful match through the pipeline. The similarity to the systolic pattern matcher in Figure 2-3 is clear.

2.2. Circuit Extensions

The syntax-directed construction procedure described above allows straightforward extension of the set of expressions that can be recognized. Cells for new operators can be used by simply adding a few productions to the grammar that describes expressions. While these new operators do not extend the class of patterns that can be recognized, they can shorten the expressions needed to describe the patterns by abbreviating commonly-used subexpressions. Since the systolic recognizer for a regular expression uses one cell for each symbol in the expression (other than parentheses), these abbreviations decrease the circuit size. These extensions are convenient in practice [1], and are used in many software tools for matching regular expressions [53].

One such cell recognizes λ, the abbreviation for φ^*. A recognizer for λ is shown in Figure 2-14, and can be used in the compiler by adding this production and semantic rule to the grammar:

$R \rightarrow \lambda$ Use a new λ cell for R.

The cell simply ties RES to ENB, so that enabled empty strings are recognized. The same effect could be achieved by allowing any port of a cell to be terminated.

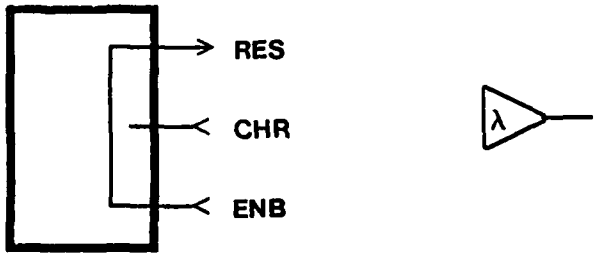


Figure 2-14: λ cell

Other common extensions are the + iterator, and the option prime. The expression E^+ is an abbreviation for $E(E^*)$, and matches 1 or more repetitions of E. The expression E' stands for $\lambda + E$, and matches 0 or 1 repetitions of E. Cells for these operators are shown in Figures 2-15 and 2-16, and can be included in circuits by adding these productions to the grammar:

$P \rightarrow (R)^+$ Connect the right port of R to the top port of a new + iterator node.

$P \rightarrow (R)'$ Connect the right port of R to the top port of a new prime node.

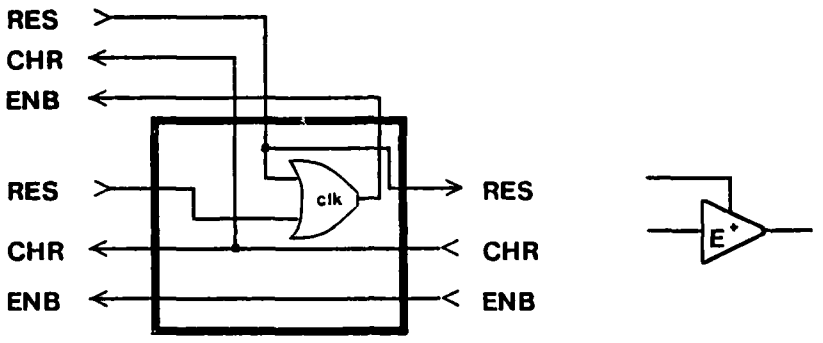


Figure 2-15: + Iterator cell

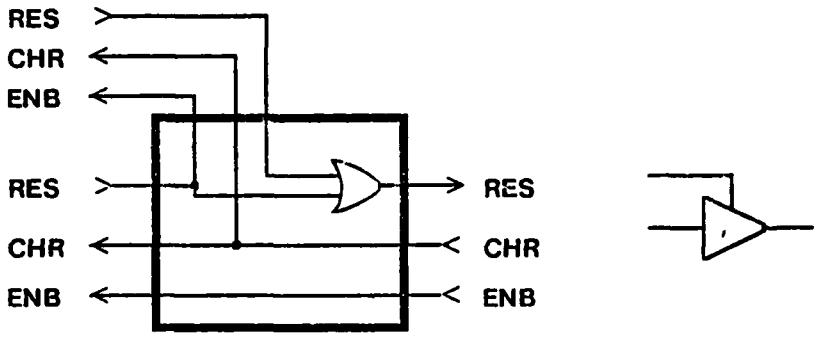


Figure 2-16: Option cell

A less common operator is the # operator, defined by the equation (a # b) ≜ a(ba)*. This is often used in the specification of programming languages to describe variable length lists of tokens separated by delimiters [73]. The # cell shown in Figure 2-17 can be used by adding the production:

$P \rightarrow (R_1 \# R_2)$ Connect the right port of R_1 to the top port of a new # node, and connect the right port of R_2 to the bottom port of the same node.

Similar cells for other operators can be designed and added in the same way.

Not all desirable operators can be added in this simple way [61]. For example, there is no cell that can be added for the intersection operation (∩) or for complementation (~). Regular expressions with these additional operators are called, respectively, *semiextended* and *extended regular*

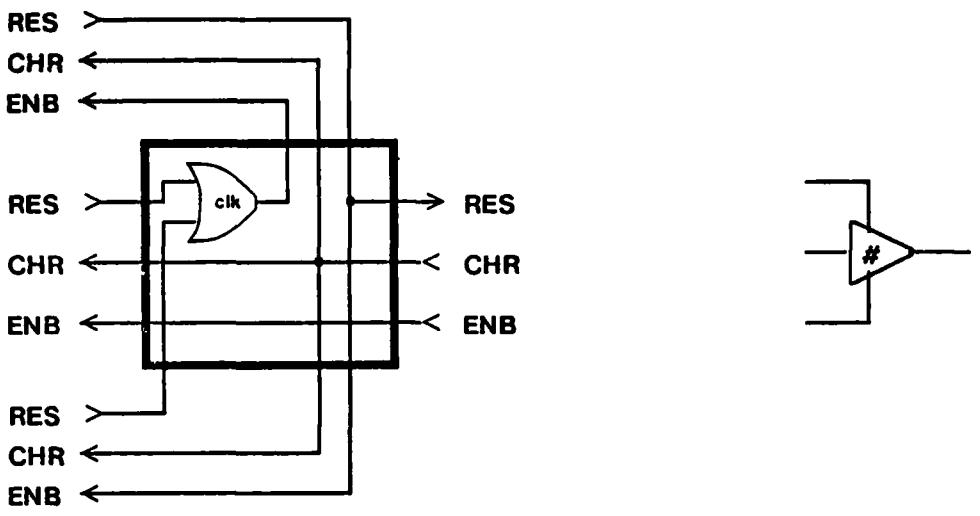


Figure 2-17: # operator cell

expressions. Hunt showed that the space required by a Turing machine to recognize a string in a semiextended or extended regular expression E is more than polynomial in the length of E [39]. There is thus no single cell that can be added to a recognizer when ∩ or ~ is encountered in parsing an expression, since this would allow a polynomial-size recognizer to be constructed, and a Turing machine could emulate the circuit using polynomial space.

A different sort of extension to regular expressions is a set of operators for describing sets of characters. For example, the extended expression {xyz}ab is a shorthand for (x+y+z)ab. This extension can eliminate some of the "+" cells in a recognizer, since the comparator cell can be modified to check CHR for membership in a set. Characters will typically be several bits wide, and compared in parallel, so one simple and useful set membership test is to allow any bit in the character to be tested for 0 or 1, or to be ignored. The comparator described in Section 3.3 has this feature. Together with operators for union, intersection, and complementation of sets of characters, such a comparator is quite powerful. For instance, the set of ASCII upper case letters can be recognized with just four comparators by using the expression

<10*aaaaa*> ∩ ~(<1000000> + <10111*aa*> + <1011011>).

The strings within angle brackets are the specifications of individual bits of the characters, where *a* indicates an ignored bit. The first string specifies the set of all characters whose first two bits are 10. This extension is easy to add and clearly useful.

Cells for intersection and complementation of sets of *characters* (as opposed to sets of *strings*) are easily constructed, and can be used by adding to the grammar the non-terminal C (standing for a set of characters) along with the following productions.

P → C	Use the circuit for C as the circuit for P.
C → <letter>	Use a new comparator node for C.
C → (C + C)	Terminate the C's and connect them to the top and bottom ports of a new + node.
C → (C ∩ C)	Terminate the C's and connect them to the top and bottom ports of a new ∩ node.
C → (~C)	Connect the C to the left port of a new ~ node.

Figures 2-18 and 2-19 show the cells for character set intersection and complementation. Note again that these cells do not operate on regular expressions. The intersection cell, for example, fails in the expression A*(B ∩ AB). This expression does not match any strings, but a recognizer built in the obvious way will recognize the string AB. This example illustrates the need for verification of specialized compilers. Before a cell for a new operator is added to the grammar, the correctness of the cell should be verified using the syntax-directed procedure of Chapt · 5.

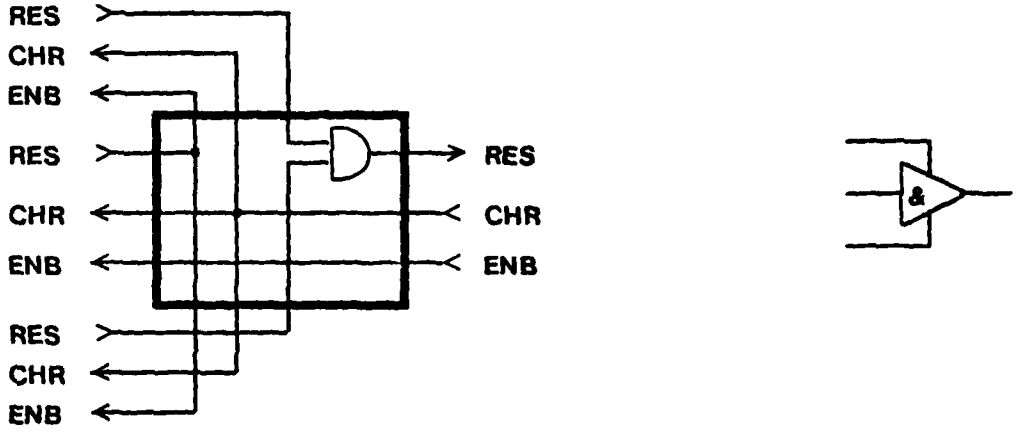


Figure 2-18: Intersection cell for sets of characters

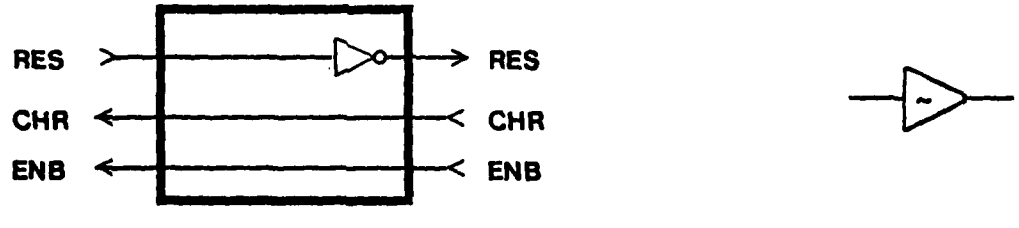


Figure 2-19: Cell for complementing sets of characters

Several application-dependent extensions to systolic recognizers are also possible. These extensions allow the recognizers to compute other values besides just the recognition or non-recognition of substrings. An appropriate choice of application-dependent extensions can convert the compiler for regular language recognition into a compiler for a more specialized domain. This increases the space and time efficiency of the final circuit, while preserving the advantages of compiling from a high-level description.

An extension with wide applicability is the addition of a disable signal to the recognizer. Such a signal would be input on the same beats as ENB is input and RES is output, and would disable matches currently in progress. If DIS_t is true, then any match whose first character is input before t and whose last character is input after t would not produce a true RES. The DIS signal simplifies the use of recognizers in lexical analysis, database filtering, and other applications in which a data stream is partitioned into tokens that match regular expressions.

The DIS signal performs the function of the initializing RESET to a recognizer, but also interacts with RES and ENB to make the recognizers more useful. Table 2-2 shows the input-output behavior of a recognizer with DIS. Any RES that is output on the same beat as DIS is input is unaffected, as is any match that is enabled on the same beat. Only partially-completed matches are disabled. Recognizers with DIS can thus be used to partition a text stream into tokens by simply feeding the RES output back into both the ENB and DIS inputs. When a substring is recognized, this feedback will start a new search while aborting any that are in progress.

Another application-dependent extension is the calculation of attribute values during the matching process. The simplest example of attribute calculation is simply the emission of RES_{out} values from cells other than the root of the tree. For example, a recognizer for ABC* could emit the RES_{out} value from the B cell as well as the "*" cell, indicating whether the matched string has any C's on the end. Ullman [76] has suggested that this technique could efficiently implement finite state controllers by using several recognizers that read and write a set of state registers. More complex attributes that could be computed during recognition include the path probabilities needed in speech recognition [55] and the generated events needed in hardware monitors [9].

Beat	ENB	DIS	CHR	RES	Comment
1	1	0		1	$\epsilon \in (ab)^*$
2			a		
3	0	0		0	
4			b		
5	0	0		1	
6			a		
7	0	0		0	
8			b		
9	0	1		1	DIS doesn't affect this beat.
10			a		
11	0	0		0	
12			b		
13	0	0		0	Disabled on beat 9.
14			a		
15	1	1		1	ENB overcomes DIS
16			a		
17	0	0		0	
18			b		
19	0	0		1	Enabled on beat 15.

Table 2-2: Trace of a recognizer with DIS for $(ab)^*$

2.3. Summary

This chapter has presented a specialized compiler for constructing systolic recognizer circuits for regular languages. The compiler uses a library of specialized cells, one for each operator that may appear in an expression. A context-free grammar for regular expressions directs the interconnection of these cells into recognizers.

The structure of the compiler, in which rules and cells are separate, has several advantages in flexibility and extensibility. For example, new circuit technologies can be used with the compiler by simply redesigning the primitive cells. No changes to the interconnection procedure are needed. The structure also permits the operators used in regular expressions to be extended systematically. Cells for new operators can be included by adding just a few lines to the grammar. The specialized circuit compiler discussed here should be a useful and flexible tool for integrated circuit designers.

Chapter 3
Layout of Systolic Recognizers

Chapter 2 describes a *circuit* compiler, outlining a syntax-directed procedure for building tree-structured recognizer circuits. A *silicon* compiler must not only construct circuits but must lay them out efficiently. This chapter describes several techniques for laying out recognizer circuits on a silicon chip.

After describing several layout schema, this chapter applies them to the design of specialized programmable layouts for language recognizers. A specialized programmable layout is used in conjunction with a specialized silicon compiler to produce chips for a limited application domain. Parts of the layout that are common to all problems in the application domain are fixed, while parts that may vary are programmable. The specialization of layouts may provide economies of scale and can decrease the design time for custom parts. Several types of specialized programmable layouts for language recognition are discussed.

The chapter ends with a description of a prototype specialized programmable layout for language recognition. This layout contains comparator cells similar to the one shown in Figure 2-5. The logic in the cells is fixed, but the characters within the cells and the interconnections between them can be programmed after fabrication. The layout was fabricated in NMOS and programmed by cutting metal lines with a laser.

3.1. Layout Schema

The silicon compilers discussed here will not produce arbitrary layouts; the layouts will fall into restricted frameworks. Section 2.1 described a technique for partial circuit design, starting from a set of pre-designed cells. Similarly, this chapter will describe several schema for partial layout design, each consisting of a layout algorithm and a floorplan. The floorplan associated with a layout scheme embodies the layout decisions that are independent of the circuit. It determines which placements of cells and data paths will be considered. The algorithm associated with a layout scheme places the

individual cells of each specific recognizer in legal positions, and routes the data paths that connect them.

Floorplans may vary in the freedom to place cells and data paths. Freedom of component placement can be traded for freedom of component design. If the positions of components are very restricted, the components themselves may be made small, though the layout algorithm might place them inefficiently. On the other hand, if the positions are unrestricted, the layout algorithm can do a good job but the components may need to be larger to allow them to fit together in more ways. If data paths are pre-placed, for instance, then the placement of cells is restricted but data paths might be packed more densely. This section examines several layout schema with floorplans of varying flexibility.

The layout algorithms in this chapter use the divide-and-conquer paradigm. A circuit to be laid out is split into two smaller subcircuits, which are independently laid out on separate parts of the floorplan. The subcircuit layouts are then interconnected, producing a layout for the entire circuit.

The crux of a divide-and-conquer algorithm is to find a method for splitting a large problem into independent subproblems, so that the solutions of the subproblems can be combined to solve the original problem. For layout of recognizers, the structure of the circuits provides a splitting method. The circuits produced by the syntax-directed procedure are bounded-degree trees. If only the nodes described in Sections 2.1 and 2.2 are used, all trees will have degree at most 3. A well-known lemma, given here without proof, provides a constant node separator for bounded-degree trees. (A proof is given by Valiant [77].) The lemma shows that by removing a single edge, one tree can be split into two trees of nearly equal size.

Lemma 3-1: In any tree T with n edges and degree r, there is an edge whose removal leaves trees T_1 and T_2 such that for some x in the range $1/r \leq x \leq (r-1)/r$,

$$|T_1| \leq xn \text{ and } |T_2| \leq (1-x)n.$$

The notation $|T|$ here means the number of edges in T.

Lemma 3-1 provides a method for dividing a recognizer circuit nearly in half by removing one data path. This method can be used with several floorplans to lay out recognizer circuits.

The most flexible floorplan allows arbitrary placement of cells and data paths. An algorithm based on Lemma 3-1 that lays out an n-node recognizer in a rectangle of area $O(n)$ was discovered independently by several researchers, including Floyd and Ullman, Leiserson, and Valiant [28, 52, 77]. Though this layout scheme makes efficient use of silicon area, it is not restructurable — different tree structures require quite different layouts.

Layouts that are restructurable have several advantages over those that are not. A restructurable layout can be used in chips that will be configured after manufacture, such as the E. T. chip described in Section 3.3. Configurable chips have advantages of economy and quick turnaround for new designs, so that restructurable layouts are worth examining. Restructurable layouts are also useful in designing chips that are not configurable. If a recognizer layout is restructurable, then the design of a chip containing a recognizer can proceed before the expression to be recognized is known. This permits separation of concerns during design of the chip and eases the correction of errors. For these reasons, this chapter will concentrate on restructurable layouts.

In restructurable layouts, the floorplan is more restricted. Cells may be placed in only certain sites and data paths are routed between them in channels. This may decrease the area-efficiency of cell and path placements, though it may permit cells and data paths to be smaller and faster.

Bhatt and Leiserson [8] have developed a restructurable layout that requires only $O(n)$ area for any n-node tree. Cells and data paths are placed in fixed locations and the layout is configured for a particular tree by programming a set of crossbar switches. For the recognizers considered here, these switches may have undesirable effects; their area may be too great and they may impose a delay on signals going through them. Design experience with the E. T. chip in Section 3.3 suggests that only a few hundred nodes will fit on a chip in the forseeable future. For such small numbers of nodes, the $O(\log^2 n)$ area taken by the crossbar switches and by the data paths that route signals between nodes and switches may be as great as the area of the nodes themselves. In addition to the area penalty for small trees, this layout can cause signals to be delayed, since some edges may pass through as many as $O(\log n)$ of the crossbar switches. In some restructuring technologies, such as the soft-programmable layouts discussed in Section 3.2, each switch may impose a delay on signals passing through it. These potential problems require the consideration of other layout methods.

Several restructurable layouts are available that use compact designs for both cells and data paths, and that require only $O(n \log^k n)$ area for laying out an n-node tree. These layouts are *collinear* — all nodes are placed on a line, and all edges are routed in channels parallel to the line. Node designs can thus be quite compact, with all ports on one edge. If two layers are available for routing, data paths can also be compact. Interconnecting two nodes simply requires connecting the ports to a common channel on one layer, and routing within the channel on the other layer. These layouts also help mitigate the problem of signal delays. Using the *cutbus* described in Section 3.2.2, layouts can be constructed in which each edge goes through only a constant number of switches.

If the only restriction imposed by a floorplan is that the layout be collinear, Rosenberg's *Diogenes* layouts [66] are the smallest known. To lay out a tree using a Diogenes layout, the tree is traversed in preorder using the wiring channels to model the stack. An n-node degree-r tree requires $\lceil r/2 \rceil \cdot \lceil \lg n \rceil$ channels[1] [16].

Another layout restriction besides collinearity may give speed advantages. In the Diogenes layouts, a data path between two ports can be routed along different channels at different points along the way. Restricting paths to stay on one channel can avoid delays, since a path that changes channels must go through a switch for each change. If no connections between channels are permitted, an algorithm due to Leiserson [52] is asymptotically optimal. This algorithm lays out a degree-r n-node tree with all edges routed in $\lceil \lg n / \lg(r/(r-1)) \rceil$ channels. Since the algorithm illustrates features common to divide-and-conquer layout algorithms, it is reproduced here as Algorithm CL.

Algorithm CL

This algorithm lays out a tree T in a line of $|T|$ or more node sites.

1. If $|T| = 1$, then place the single node in a node site and return. Otherwise, follow the remaining steps.
2. Using Lemma 3-1, remove edge E to split T into two trees T_1 and T_2 of about equal size.
3. Split the $|T|$ sites into two blocks of $|T_1|$ and $|T_2|$ sites.
4. Lay out T_1 and T_2 in their respective blocks.
5. Route edge E using a segment of a single channel that extends the length of the line of nodes.

Figure 3-1 shows the operation of this algorithm on a small tree. The separator edge is highlighted at each step and the assignments of subtrees to blocks of nodes are shown. The routing channels are split where they cross the boundaries of a block of nodes so that several edges can share a channel.

The layout area of an n-cell recognizer using this algorithm is $O(n \log n)$. Although collinear layouts of some recognizers can be built with linear area, there are trees whose collinear layouts require $n \log n$ area [11]. As long as layouts are to be collinear, then, with arbitrary placement of cells on the line and arbitrary division of channels, this divide-and-conquer algorithm is asymptotically optimal.

---

[1] The symbol lg n stands for $\log_2 n$

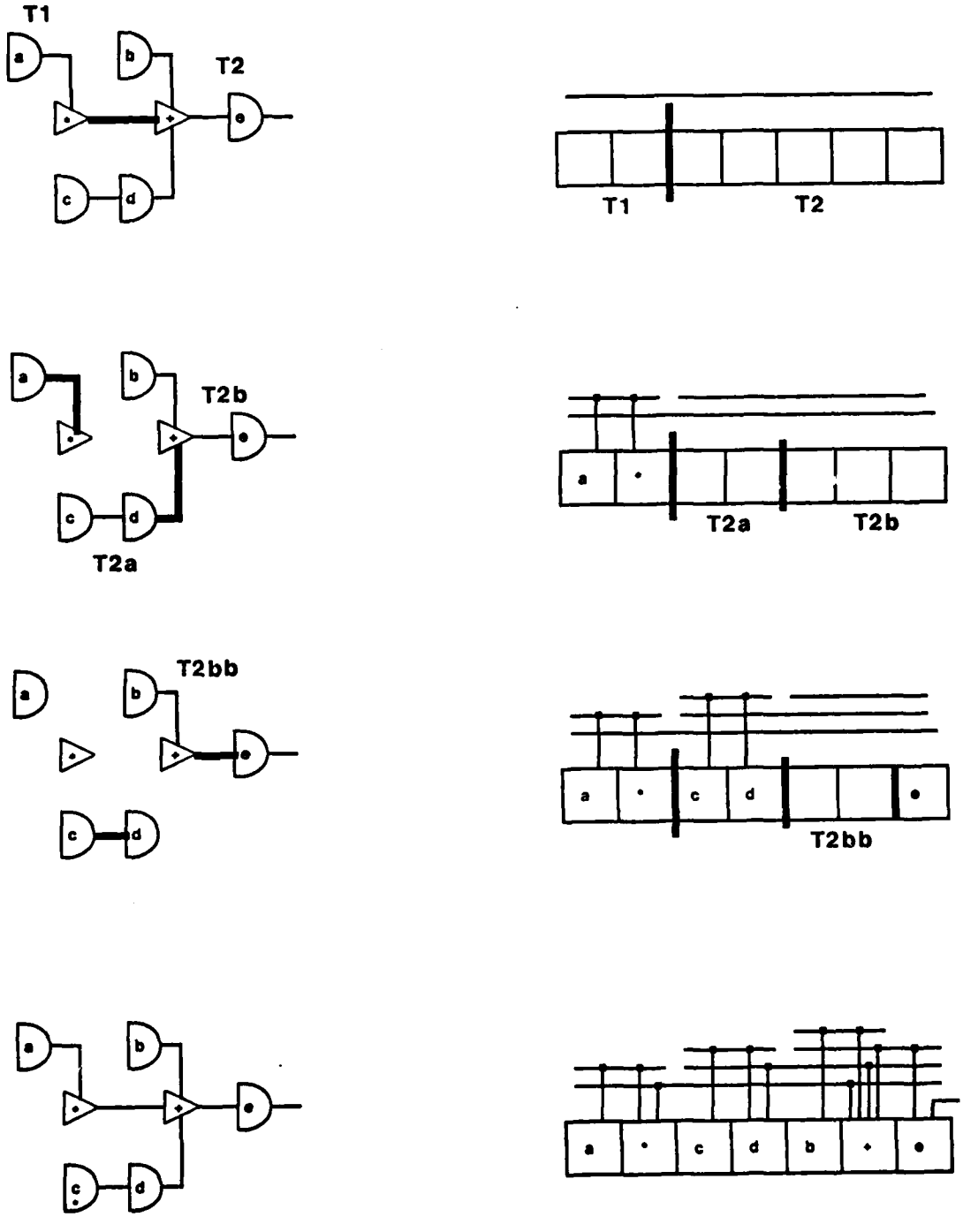


Figure 3-1: Collinear layout of a recognizer for a*(b+cd)e

An even more restricted layout scheme may be useful. The nodes used in constructing recognizers can be divided into two types:

- Comparators, which have few ports, but many gates;
- Combinators, which have many ports, but only one or two gates.

It may be worthwhile to reserve node sites for one or the other of these types of nodes. A collinear layout with only $O(n \log^2 n)$ channels can be constructed, even if node sites are reserved. The layout depends upon a lemma of Bhatt and Leiserson [8].

> Lemma 3-2: Let T be a tree with n edges and degree r, and with nodes of two colors, black and white. Then there is a set of $2 \cdot \lceil \lg n/\lg(r/(r-1)) \rceil$ edges whose removal divides the set of nodes, the set of white nodes, and the set of black nodes in half. Neither set then has more than one more node, black node, or white node than the other set.

Lemma 3-2 permits construction of Algorithm TCL, a divide-and-conquer layout algorithm similar to Algorithm CL. This two-color layout algorithm lays out a degree-r n-node recognizer on reserved node sites, using at most $2 \cdot \lceil \lg n \rceil \cdot \lceil \lg n/\lg(r/(r-1)) \rceil$ channels.

Algorithm TCL

This algorithm lays out a recognizer T with $w(T)$ comparators on a line of $|T|$ node sites with $w(T)$ sites for comparators evenly distributed along the line. Even distribution requires that any contiguous line of $\lfloor |T|/w(T) \rfloor$ node sites contain at most one site for a comparator, and that any contiguous line of $\lceil |T|/w(T) \rceil$ sites contain at least one comparator site.

1. Using Lemma 3-2, remove a set S of $2 \cdot \lceil \lg n/\lg(r/(r-1)) \rceil$ edges to split T into recognizers T_1 and T_2 with $|T_1| = \lfloor |T|/2 \rfloor$, and $w(T_1) = \lfloor w(T)/2 \rfloor$.
2. Split the line of nodes into blocks of $|T_1|$ and $|T_2|$ node sites, with $w(T_1)$ and $w(T_2)$ comparator sites respectively.
3. Lay out T_1 and T_2 in their lines of sites.
4. Route the set S of edges, using $2 \cdot \lceil \lg n/\lg(r/(r-1)) \rceil$ segments that extend the length of T.

This section has presented layout schema of asymptotically optimal area for both reconfigurable and non-reconfigurable layouts, under several kinds of restrictions. By selecting among these layout schema, a specialized silicon compiler can construct area-efficient layouts for any class of recognizer circuits. Starting with a regular expression, then, the compiler can construct and lay out an efficient systolic language recognizer.

3.2. Programmable Layouts for Recognizers

The value of specialized silicon compilers increases if they are used in conjunction with *programmable layouts*. A programmable layout is an incompletely specified layout for a circuit in the compiler's application area. A complete specification of the circuit to be built can be translated into a program, or complete specification, for the layout. A programmable layout thus serves as a target architecture for the specialized silicon compiler. Programmable layouts have several advantages.

- Economy: a large number of chips can be fabricated before they are needed. Individual chips can then be programmed as need arises.
- Predictability: performance, area, and power requirements of chips can be well characterized before they are programmed. Variations due to design and fabrication are minimized.
- Quick turnaround: configuring a programmable chip is faster than designing and fabricating a custom layout. Maskmaking and etching steps are eliminated, and chips can often be configured after packaging.

Designers and manufacturers have long recognized these advantages, and have built such programmable layouts as read-only memories, programmed array logic, and single-chip microprocessors [13]. This chapter introduces a programmable layout, called the *programmable recognizer array* (PRA), for language recognizers. Several implementations are discussed, and the design and testing of a prototype programmable recognizer are presented.

The restructurable layouts for recognizers discussed in Section 3.1 can be used as programmable layouts. The placement of cells and data paths can be determined before the details of the tree structure are known. The pre-determined positions for cells and wires can be considered as programmable areas in these layouts; a layout is programmed by placing the correct structures in those areas. Details of programmable recognizers are considered in this section.

The layouts considered in this section use a standard floorplan based on the collinear layouts discussed in Section 3.1. Cells are placed in a line and routing channels run along the line of cells. These layouts are the most practical in today's technologies, since the crosspoint switches needed for the linear area layout may require too much area and may impose excessive delays in some cases.

The basic floorplan shown in Figure 3-2 can be used for all programmable layouts in this chapter. Bonding pads are arrayed around the edge of the chip. Inside the ring of bonding pads are one or two rows of cells, with routing channels running along the rows. (Although two rows of cells are

shown in the figure, only one row might be used if the cells were larger.) Both cells and channels are programmable. As the number of devices that can be fit on a chip increases, this floorplan may be extended by adding additional rows of cells, with routing channels between them.

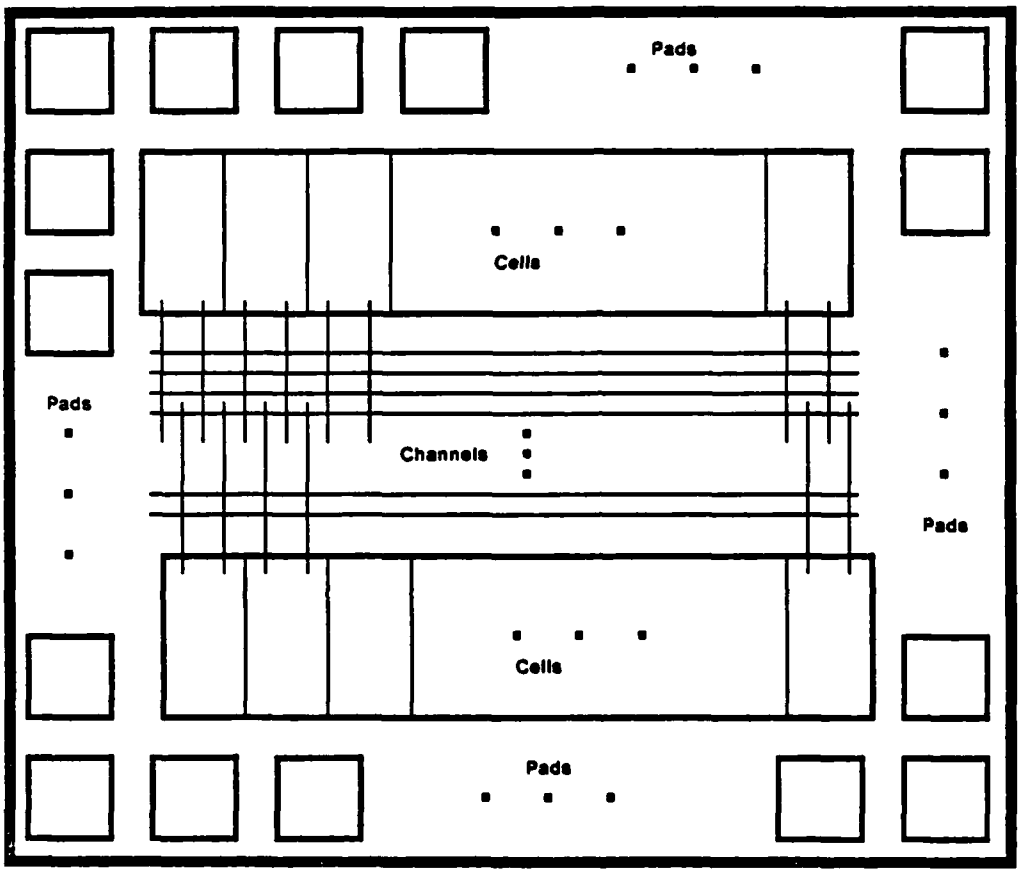


Figure 3-2: Common floorplan for programmable recognizer layouts

This section discusses three kinds of programmable layouts:

- Mask programmable layouts, which can be programmed by changing one or two masks during fabrication;
- Fusible link layouts, which are programmed after fabrication by making or breaking connections (with a laser, for example);
- Soft programmable layouts, which are programmed (and re-programmed) by setting switches (such as pass transistors) that make or break connections.

These types of layouts have different applications, and choice of one of them depends on the

anticipated use. The problem is analogous to choosing between ROM, PROM, and RAM in a memory application. Mask programmable layouts are superior for applications in which a large number of chips will be permanently configured the same way. The cost of producing the masks is high, but the chips are likely to make efficient use of area and be quite fast. Fusible link layouts are superior when chips will be permanently configured, but only a few will have each program. The fusible links take up extra area on the chip and may decrease its speed. Soft programmable layouts are ideal for recognizers that may be reused for different expressions. Though a soft programmable layout is very flexible, it takes more space than either of the other two types.

The recognizers discussed in Chapter 2 are made up of two kinds of components: cells to do the computation, and data paths to transmit data between cells. A programmable layout for a class of recognizers must provide both configurable cells and configurable channels. The next two sections discuss techniques that can be used with the floorplan in Figure 3-2.

3.2.1. Programmable Cells

Individual cells in a layout must be programmable for different functions. Comparators, for example, should be programmable to recognize any character. In fact, they should be somewhat more flexible than this, since regular expressions that occur in practice often have subexpressions that match large sets of single characters. Programs such as LEX [53], which allow a user to specify patterns with regular expressions, nearly always include operators for sets of single characters. Programmable layouts can provide analogous operators, since comparator cells can match sets of characters instead of just single characters. For example, a comparator could be programmable for a *wild-card* character that matches any character whatsoever. More flexibly, individual bits of a pattern character could be programmable as wild-card bits that match either 0 or 1. If characters were represented in ASCII, for instance, wild-card bits would allow a single comparator to match any control character.

The similarity of the combinator cells to each other mandates that they too should be programmable. Each of the cells discussed in Chapter 2 uses a single gate to compute its ENB and RES outputs, and simply transmits CIIR unchanged from input to output. The clocked OR gate used in the closure cells can be used without change in the other combinators, so that only a few connections need be changed to program a cell for a particular combinator.

The combinator cells should be modified slightly if the floorplan of Figure 3-2 is used. The change, which is a rerouting of the CIIR signal, can save chip area. In the modified routing, CIIR is routed

entirely within the routing channels above the cell so that CHR does not enter the combinators. Instead, the CHR lines on the channel that is connected to the right port of the cell are wired directly to the channels that are connected to the other ports of the cell. The CHR signal can then be omitted from the ports of combinators. This can save a significant amount of space, since a cell is at least as wide as the ports that enter it. The CHR signal is likely to be several bits wide, so that removing it from three or four ports decreases the width of the combinator. Figure 3-3 shows a layout of a small recognizer using the original routing scheme in which the CHR signal is routed through the combinators. Figure 3-4 is a layout of the same recognizer using the modified CHR routing. The portions of the channels that are actually used in routing are indicated by heavy black lines in the figures, while unused portions are indicated by lighter lines. Notice that the combinator in Figure 3-4 can be much narrower than that in Figure 3-3, because fewer lines must cross the upper edge.

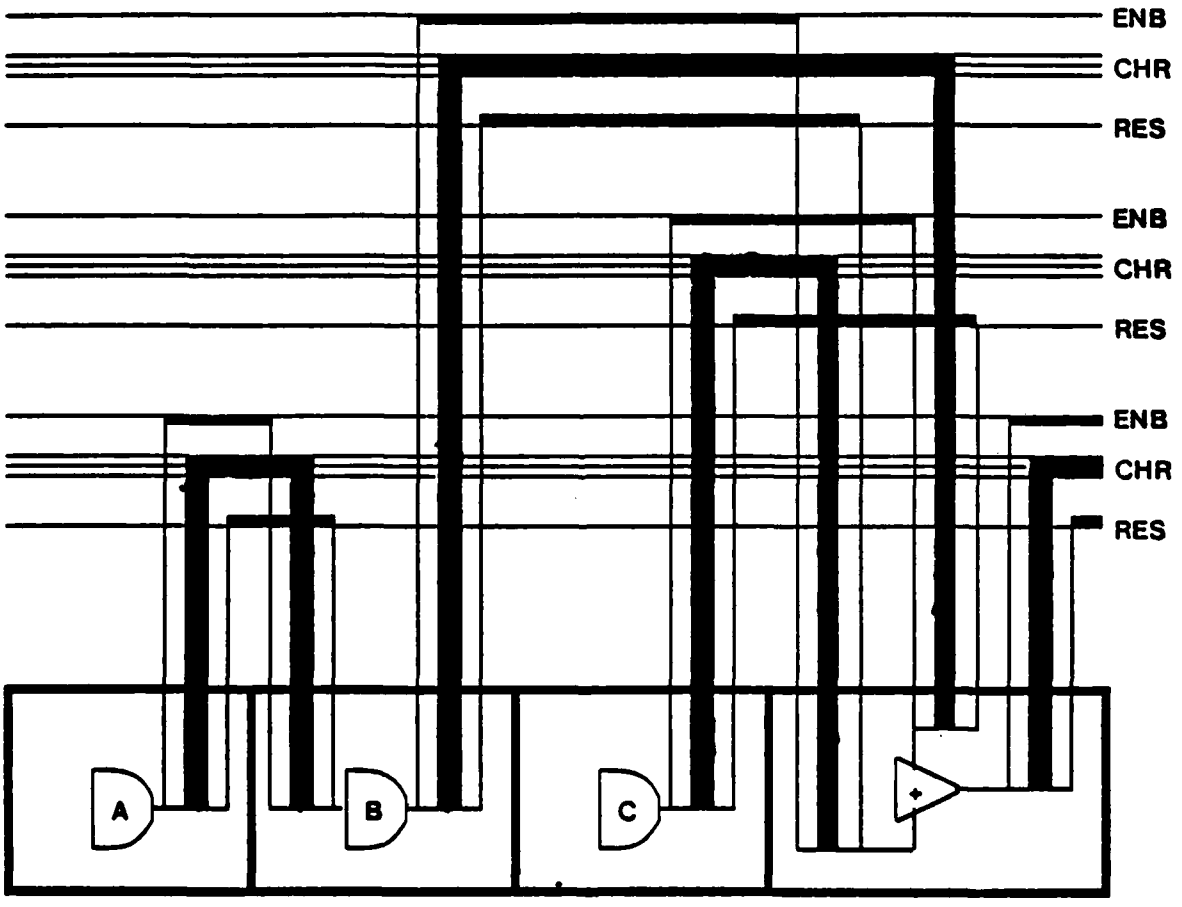


Figure 3-3: Original CHR routing in a layout for AB + C

Both comparator and combinator cells should be programmable. Should each cell in the layout be

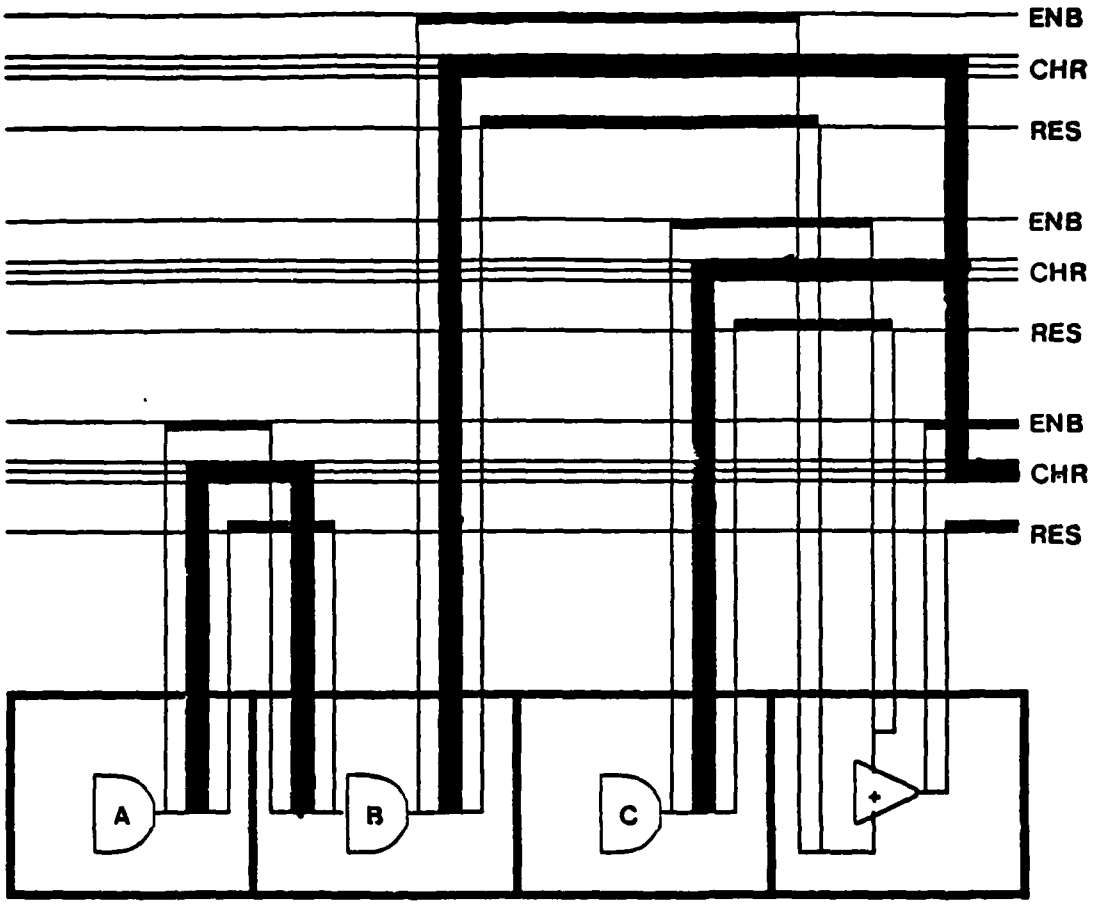


Figure 3-4: Modified CHR routing in a layout for AB + C

programmable as any type of cell, or should the cells be divided into types before programming? The alternative that maximizes the number of usable cells that fit on a single chip should be chosen.

The number of cells that fit on a chip with the floorplan of Figure 3-2 depends upon the number that can fit in each line of cells. All cells in the floorplan are the same height, but may differ in width depending upon the number of ports entering each one. The number of useful cells that can fit in a line thus depends upon the utilization of the ports. It is profitable to think of programming the cells as programming the ports, in order to group them into cells. Several techniques for programming the ports to form cells can be imagined.

1. Each single port might be programmable as any port of any combinator or comparator. Then two adjacent ports would need to be programmed to make a comparator, or three to make a Kleene * node.

2. Groups of four ports could be assigned to a cell that was programmable as any single comparator or combinator.

3. Ports could be split into groups of two and four, where groups of two would be programmable as comparators, and groups of four as combinators.

4. A set of wires, programmable as either two *full* ports (with CIIR) or four *modified* ports (without CIIR) could enter each cell. Each cell could then be programmed as any comparator or any combinator.

Techniques 1 and 2 can be ruled out. Technique 1 requires a large, complex layout for each port to allow it to emulate the eight different types of ports on cells. Technique 2 wastes half of the ports for each comparator. Since comparators make up a substantial fraction of the cells in a recognizer, this waste of area is unacceptable.

Techniques 3 and 4, on the other hand, merit consideration. Technique 3 is a division of cells into two types. Algorithm TCL can be used to lay out a recognizer in a line of typed cells. If the relative frequency of comparators and combinators can be predicted in advance, this technique probably gives the smallest cells.

If the frequencies of cell types cannot be predicted, however, technique 4 is probably better. The relatively small amount of logic needed in a combinator could be added to a comparator without an appreciable blowup of area. If characters were five to ten bits wide, the number of wires needed by a single port with CIIR would be about the same as needed by two or three modified ports, without CIIR. Thus, neither the width nor the area of cells would increase under this technique, and two-color layout algorithms are not needed.

In any of these techniques, the functions of combinators should be expanded to make effective use of silicon area. The logic in the OR and Kleene * combinators takes up considerably less space than the logic in a comparator, yet the area in the floorplan that is allocated to a combinator is comparable to that of a comparator. Most of the space in a programmable combinator is therefore wasted. This space can be used more effectively by increasing the set of operators that can be programmed into a single combinator. Common operators such as the Kleene +, indicating repetition one or more times, or the prime ('), indicating an optional subexpression, can be added easily. The combinators could also be programmable for the set operators for fixed-length strings mentioned in Section 2.2, such as intersection and complement. This option would interact well with the wild-card bits in the programmable comparators, to allow recognition of arbitrary sets of characters using very few cells. A well-chosen set of operators can decrease the size of practical programmable layouts.

3.2.2. Programmable Channels

A programmable set of wiring channels must allow ports to be interconnected as specified in Chapter 2. It must be possible to connect any port to any one of several channels, and non-overlapping connections between pairs of ports must be able to share a channel without interference. In the floorplan of Figure 3-2, the ports are connected to wires that run perpendicular to the channels. If programmable connections are placed where the ports cross the channels, any port can be connected to any of the channels.

To allow non-overlapping connections to share a channel, the channels must be split so that the connections are electrically isolated. Programmable *cutpoints* can be placed along the channel at points at which it may need to be split. It certainly suffices to place a cutpoint between each pair of ports on a channel, though fewer cutpoints may suffice for some channels.

One approach to designing the programmable channels is to design a *crossing point* to be used where ports cross channels. This crossing point allows the port to be connected to the channel and allows the channel to be split on one side of the port. Figure 3-5 shows the conceptual design of a crossing point, with optional connections denoted by thinner lines. A port or channel will usually consist of more than one wire, so that each line in Figure 3-5 corresponds to several wires. In programming the layout, the optional connections can be set to connect the port to the channel or to split the channel to the left of the port.

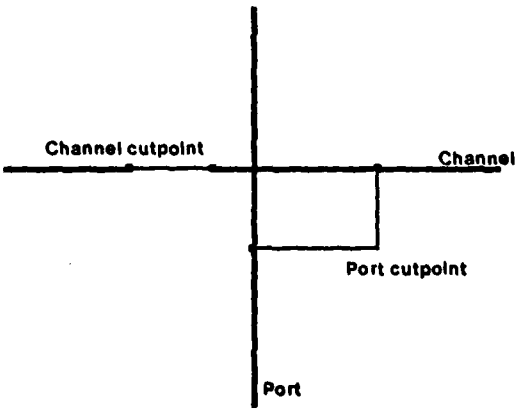


Figure 3-5: Conceptual crossing point in a programmable layout

This crossing point is easy to implement in all three types of programmable layouts. In all

implementations, the channel can be routed on one conducting layer, say metal, and the port can be routed on another, such as polysilicon. The design of the optional connections varies.

- In a mask programmable layout, the desired connections can be written on the mask and those not desired can be omitted. Only one or two mask steps are needed for this implementation.

- In a fusible link layout, the optional connections can be fusible links. Undesired links can be broken.

- In a soft programmable layout, optional connections can be MOSFET's used as pass transistors. The gate of the pass transistor is set high to make the connection or low to break it. A register on the chip holds the state of each optional connection.

The mask programmable and fusible link layouts are simple to implement. By contrast, the soft programmable layout presents some problems. Not only do the pass transistors and registers take up space on the chip, but sending signals through many pass transistors may substantially decrease the performance of the chip. In typical NMOS processes, for example, each pass transistor adds the equivalent of about $10^4\Omega$ of resistance to the line passing through it [57]. Gate inputs are largely capacitive in NMOS, so the added resistance slows down the signal propagation within the channel. For example, simulations using SPICE [25] show that propagation from a large push-pull driver to an inverter input through 2mm of polysilicon is slowed from 5ns to 70ns by the addition of 20 pass transistors. If every crossing point has a cutpoint built with a pass transistor, then an edge that crosses n ports must pass through $n+2$ transistors as shown in Figure 3-6. Since collinear layouts of n-node trees may contain edges as long as $\Omega(n)$ [11], the performance of tree recognizers will be limited by the parasitic delays in the programmable channel. These delays cause some problems in recognizers on single chips, but will be even more important if wafer-scale integration is used. If a recognizer is as large as an entire wafer, some signals may go through hundreds of crossing points. In both current and future technologies, parasitic delays must be decreased.

The solution to this performance problem in soft-programmable layouts is to use fewer cutpoints. Some of the programmable channels are reserved for long edges, and others are used for short edges. Placing fewer cutpoints in the long-edge channels ensures that all edges go through only a small number of pass transistors. Ideally, cutpoints in a channel would be placed so that no edge in that channel passed through any cutpoints. In that case, they could be breaks in conducting lines rather than transistors. Every edge would then go through only two transistors, as shown in Figure 3-7.

Figure 3-8 shows a *cutbus*, which reserves long segments for long edges and short segments for short edges. In a cutbus, the channels are divided into groups based on the number of cuts in the

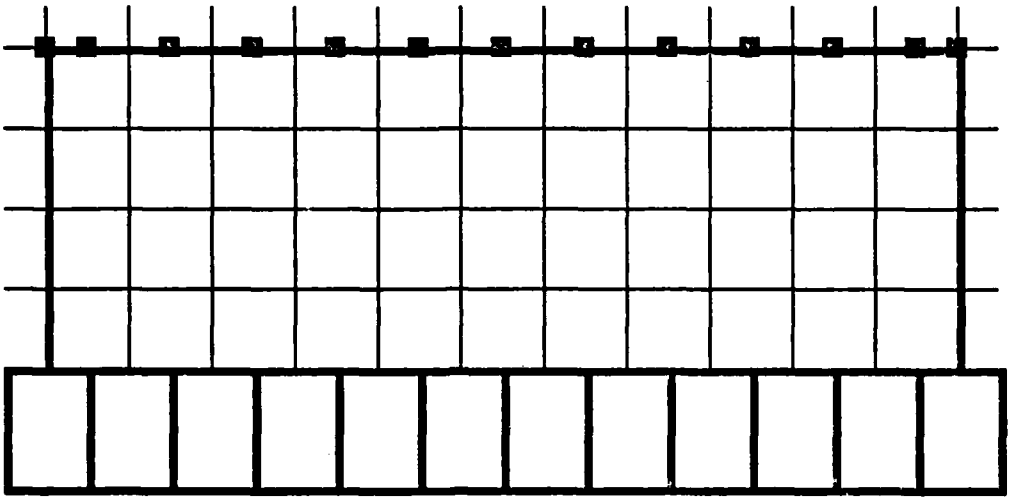


Figure 3-6: A long edge crosses $n+2$ transistors (black squares)

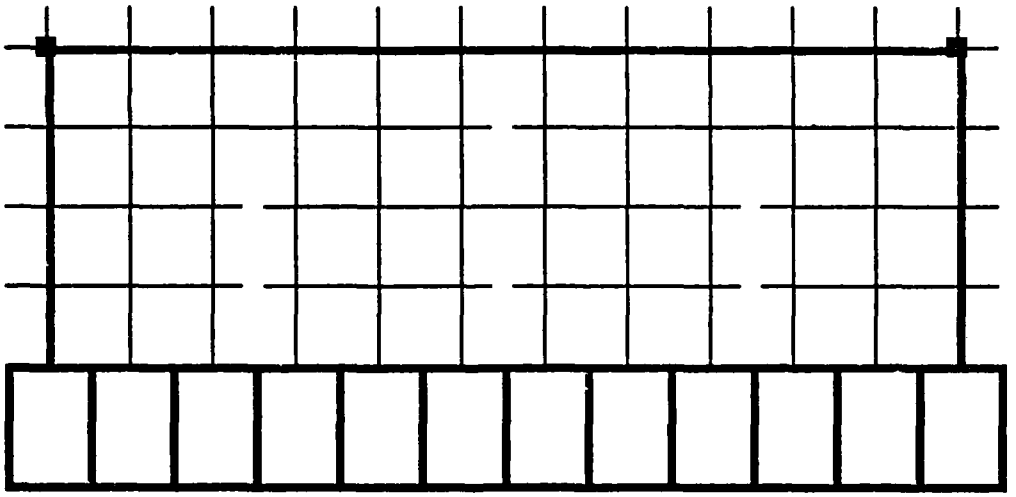


Figure 3-7: A long segment with no cutpoints.

channel. Group 0 has no cuts and is used for the very longest edges; each channel in group 0 can carry one edge of arbitrary length. Channels in group 1 have one cut and can carry two independent edges (one in each segment). The number of cuts per group increases, until a group is reached whose segments are just long enough to carry an edge between adjacent cells. Using this cutbus, a tree can be laid out so that each edge passes through a small constant number of cutpoints.

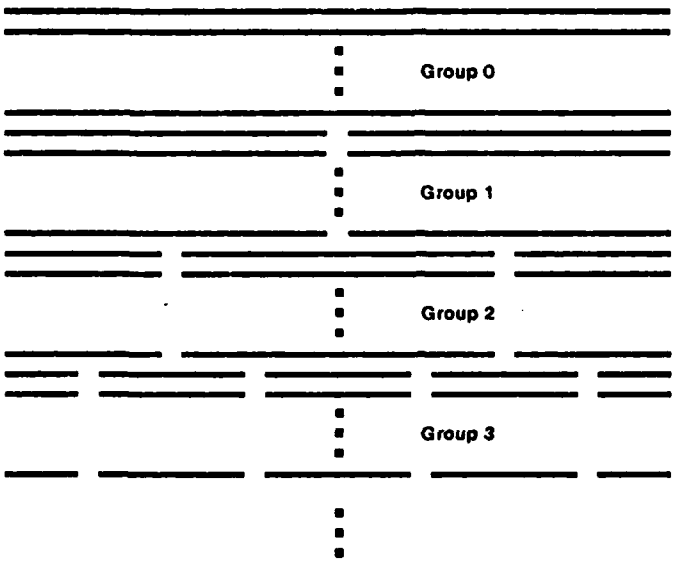


Figure 3-8: One design for a cutbus

An apparent drawback of the cutbus is that the number of routing channels may increase, since the number of edges that can be routed in each channel is restricted. Despite appearances, however, the blowup is only a small constant factor. Any degree r tree with n nodes can be laid out using only $3 \cdot \lceil \lg \ n \rceil \cdot \lceil 1/\lg \ (r/(r-1)) \rceil$ channels, even if no edge is permitted to pass through any cutpoints. The idea is to start with Algorithm CL for collinear layout, described in Section 3.1, and show that only a constant factor blow-up in channels is needed to route the edges in the cutbus.

In Algorithm CL, the tree T is divided into two nearly equal subtrees T_1 and T_2 by removing one edge. T_1 is then laid out on the left half of the line, T_2 is laid out on the other half, and the edge is routed in one of the channels. We call the consecutive groups of nodes that are used for layout of subtrees *assigned blocks*, or simply *blocks*. Thus, at the beginning of the algorithm, there is a single assigned block for the whole tree. Each stage of the algorithm divides every assigned block into two blocks. The assigned blocks that are formed during layout thus form a rooted tree, where the sons of a block are the two blocks that are formed from it. Assigned blocks are disjoint, unless one is a descendent of the other (in which case the descendent is contained in the ancestor).

Each edge of the tree being laid out corresponds to one assigned block, linking its halves into one block. Edges are routed in *channels*, and each channel may contain several cutpoints that split it into

disjoint *segments*. Notice that no edge is longer than its corresponding assigned block, so that if a segment contains both endpoints of an assigned block, it can be used to route that block. These definitions are illustrated in Figure 3-9. The edge shown can be routed in the single segment of the upper channel, to combine the two assigned blocks into one.

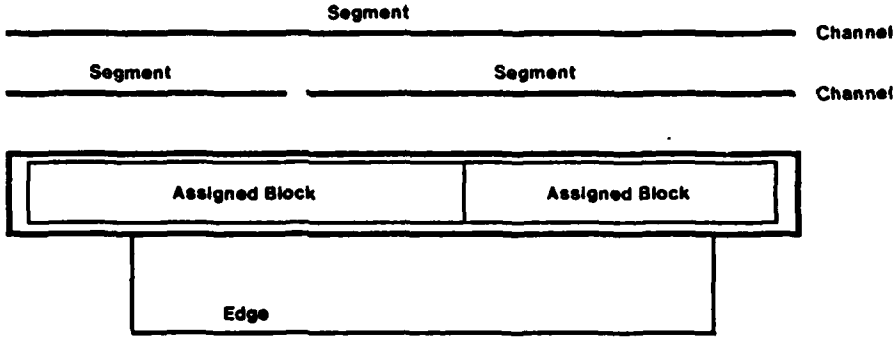


Figure 3-9: Definitions of terms

To bound the number of channels needed to lay out a tree of size n and degree r, we first prove that a set of disjoint blocks that are all about the same size can be routed in 3 channels. We then show that the assigned blocks that are formed during the layout of any tree can be split into $\lceil \lg\ n \rceil \cdot \lceil 1/\lg\ (r/(r-1)) \rceil$ such sets. Then, since each set is laid out in 3 channels, only $3 \cdot \lceil \lg\ n \rceil \cdot \lceil 1/\lg\ (r/(r-1)) \rceil$ channels are needed overall.

Lemma 3-3: For any natural numbers n and k, there is a set of 3 channels with segments of size $3k$ that suffices to route any set of edges between n collinear nodes, as long as the edges have the following properties.

- For some k, every edge crosses more than k nodes, but no more than $2k$ nodes.
- No two edges cross the same node.

Proof: The segments in the three channels have different offsets with respect to the left end of the line of nodes, as shown in Figure 3-10. One set of edges is aligned with the left end, a second is shifted right by k nodes, and the third is shifted right by $2k$ nodes.

A routing in a set of segments is an injective function from the edges to the segments, where each edge is assigned a segment that contains its endpoints. Since the function is injective, no segment is assigned more than one edge. One such injection maps each edge E into the unique segment S such that the left end of E is contained in S, and is less than k nodes from the left end of S. In other words, E is mapped to S iff the left end of E is within the leftmost k nodes of S

This is a function, since each point on the line is within the leftmost k nodes of precisely one segment.

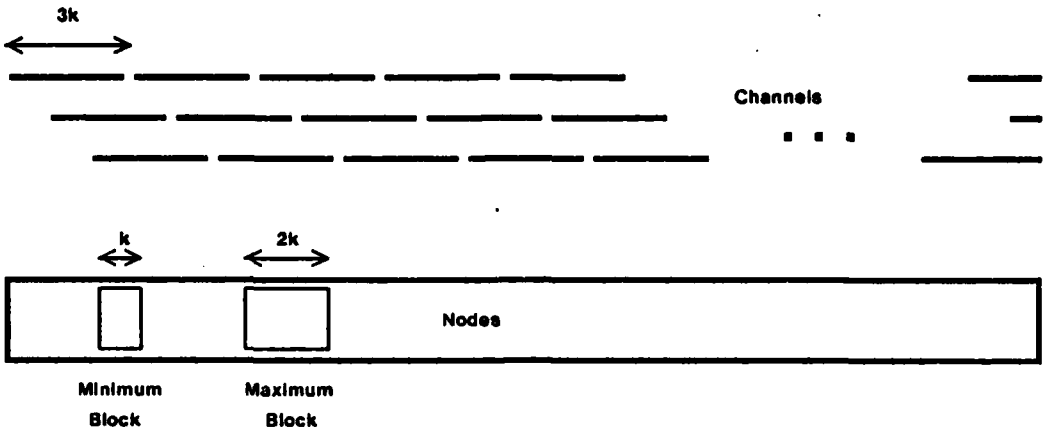


Figure 3-10: Offsets of segments in 3 channels

Edge E will fit within segment S, since the right end of E is within $2k$ of its left end. Either the right end of S is at least $2k$ away from the left end of E, or the right end of S is also the right end of the line of nodes.

Since each edge is more than k long, and the edges are disjoint, no two edges have their left ends within k of each other. Thus, no two edges are mapped to the same segment, so the function is injective.

Since the function finds a segment that contains each edge, and routes only one edge in any segment, it routes the entire set of edges without conflict.

□

This lemma can be used to show that no more than $3 \cdot \lceil \lg \ n \rceil \cdot \lceil 1/\lg \ (r/(r-1)) \rceil$ channels are needed to route the edges for any tree, even if the cutpoints are positioned in advance. This shows that the cutbus can be used for tree layout on soft-programmable chips.

Theorem 3-4: For any integers n and r, a set of $3 \cdot \lceil \lg \ n \rceil \cdot \lceil 1/\lg \ (r/(r-1)) \rceil$ channels that are split into segments can be constructed so that any degree-r tree with n nodes has a collinear layout in which the edges are routed using the segments (without additional cuts).

Proof: The channels should be divided into $\lceil \lg \ n \rceil$ sets, where the segments in each set of channels are all the same length. For $i>1$, the segments in set i are $3 \cdot 2^{-i} n$ nodes long, and those in set 1 are n nodes long. Each of the sets contains $\lceil 1/\lg \ (r/(r-1)) \rceil$ groups of three channels, where the segments in the three channels in a group are offset as in Lemma 3-3. This set of $3 \cdot \lceil \lg \ n \rceil \cdot \lceil 1/\lg \ (r/(r-1)) \rceil$ channels suffices to route the tree edges.

Use Algorithm CL to place the nodes, and to construct assignable blocks. No edge is longer than the assignable block that it forms, so that it suffices to show that we can route a set of edges, each as long as one of the assignable blocks. Thus, rather than route the edge shown in Figure 3-9, we would route an edge that is as long as the upper segment.

As noted above, the assignable blocks form a rooted tree, where the sons of a block are the two blocks into which it is split by the algorithm. We will split this tree into $\lceil \lg n \rceil$ subforests, each of which can be routed using $3\lceil 1/\lg (r/(r-1))\rceil$ channels. This will prove the theorem.

Subforest i is the set of blocks with sizes in the interval $(2^{-i}n, 2^{1-i}n]$. There are $\lceil \lg n \rceil$ of these subforests. Each subforest can be divided into a number of "levels", where each level consists of non-overlapping blocks. The first level in subforest i is the set of roots of subtrees in the subforest, *i.e.* blocks whose fathers are larger than $2^{1-i}n$, and are therefore not in the forest. The j'th level is the set of sons of $j-1$'th level blocks that are not yet too small. There are at most $\lceil 1/(\lg (r/(r-1)))\rceil$ levels, since Lemma 3-1 shows that the blocks on each level decrease in size by a factor of at least $r/(r-1)$.

To see that no two blocks in a level overlap, recall that two assigned blocks overlap only if one is a descendent of the other. Thus, none of the blocks on level 1 overlap, since each of them has no ancestor in the forest. Similarly, the blocks on level j cannot overlap, since their fathers are all on the $j-1$'th level. Thus, each level consists of non-overlapping blocks with sizes in the interval $(2^{-i}n, 2^{1-i}n]$. By Lemma 3-3, each level can be routed using three channels with segments of size $3\cdot 2^{-i}n$. Each subforest can thus be routed using $3\lceil 1/\lg (r/(r-1))\rceil$ channels. Since there are $\lg n$ subforests, a total of $3\cdot\lceil \lg n\rceil\cdot\lceil 1/\lg (r/(r-1))\rceil$ channels are needed for the whole tree.

□

Theorem 3-4 shows that, up to a small constant factor, no more channels are required in the worst case for the cutbus than are required if Algorithm CL is used with cutpoints at all points in the routing array. Moreover, the wire lengths do not increase by more than a constant factor. This indicates that soft programmable recognizers can be built without the speed penalties imposed by long chains of pass transistors. The construction of Theorem 3-4 requires that any node be able to fit in any node site, however, since Algorithm CL is used for placement. A second theorem will show that two-color placement using Algorithm TCL is also possible in a cutbus, so that some node sites can be reserved for comparators, and the rest for combinators. The number of channels needed for this cutbus layout is the same as that needed by algorithm TCL: $2\cdot\lceil \lg n\rceil\cdot\lceil \lg n/\lg (r/(r-1))\rceil$

Theorem 3-5: For any integers n and r, a set of $2\cdot\lceil \lg n\rceil\cdot\lceil \lg n/\lg (r/(r-1))\rceil$ channels that are split into segments can be constructed so that any degree-r tree with n nodes of two colors, white and black, has a collinear two-color layout in which the edges are routed using the segments with no additional cuts, and the white nodes are distributed evenly along the line.

Proof: The set of channels is divided into $\lceil \lg n\rceil$ groups, each group containing $2\cdot\lceil \lg n/\lg (r/(r-1))\rceil$ channels. The channels in each group are divided identically into segments. Channels in group 0 contain 1 segment each, spanning the n nodes. Channels in group 1 contain 2 segments, each spanning $n/2$ nodes. Channels in group i contain 2^i segments, each spanning $n/2^i$ nodes.

Algorithm TCL can be used to place nodes to be routed using this cutbus. Algorithm

TCL is recursive, with depth $\lceil \lg n \rceil$. At each call, the tree T is divided exactly in half, along with the set of white nodes, by removing $2 \cdot \lceil \lg n / \lg (r/(r-1)) \rceil$ edges, then routing those edges in a set of segments that extend the length of the tree. The segments needed at depth i are no longer than half the length of the segments needed at depth $i-1$. At depth 1, then, one set of segments of length n is needed. At depth 2, two sets of segments, each of length $n/2$ are needed At depth i, 2^i segments, each of length $n/2^i$ are needed. This is exactly what is provided in the cutbus above, so this cutbus accommodates the routing needed by Algorithm TCL.

□

Theorems 3-4 and 3-5 show that soft programmable layouts can be built that avoid the delay imposed by long chains of pass transistors. Cutbusses with few cutpoints can be built using these theorems that are only a small constant factor larger than programmable layouts with cutpoints between every pair of cells. All three styles of configurable layout are thus feasible for use in programmable recognizer layouts.

3.2.3. Placement and Routing

Constructing a programmable recognizer array is only half of the job. A technique for programming the array for any regular expression is also needed. Since our recognizer circuits are tree structured, the programming problem comes down to embedding the tree within the array.

Although Theorems 3-4 and 3-5 show the existence of programmable layouts that allow the embedding of any tree with n nodes, it may be advantageous to use fewer channels than are required in the most general case. Many of the trees corresponding to regular expressions are long and leggy, rather than bushy. It has been estimated [34] that over 90% of the state transitions in regular languages in applications correspond to simple concatenations. In tree structured recognizers, these simple concatenations become long chains of short edges, which can be laid out using only one or two channels in the array. If recognizers use only a few channels then programmable layouts should have only a few channels. It makes no sense to supply routing channels that are never needed.

Using fewer than the maximum number of channels creates a problem, however. The simple divide-and-conquer algorithm for collinear tree layouts may no longer do the job. A better tree layout scheme is needed, incorporating placement of the nodes and routing of the edges between them.

Because the design of programmable channels differs in the soft programmable layouts from the fusible link and mask programmable layouts, the placement and routing schemes will differ as well.

As in channel design, the placement and routing problems are harder in the soft programmable layout than in the other two.

In the fusible link and mask programmable layouts, the tree embedding problem is similar to the ordinary planar layout problem. A tree can be embedded in a programmable layout if and only if the cutwidth of the tree is smaller than the number of available channels. Since the min-cut layout of a tree can be determined in polynomial time [81], an embedding can be found quickly if one exists. Cutbus layout, on the other hand, which is needed for the soft programmable layout, is a new problem.

In a cutbus routing is simpler than placement. Placement consists of assigning nodes to positions on the line, while routing consists of assigning edges to segments. If placement is done first, routing becomes simply matching in a bipartite graph, which has a polynomial time solution [47]. Here, one set of nodes is the set of tree edges, the other set is the set of segments, and an edge from a tree edge to a segment means that the edge can be routed in the segment. The real problem in the soft programmable layout is placement.

The placement problem for laying out an arbitrary tree in an arbitrary cutbus is NP-complete, since the graph bandwidth problem [32] can be reduced to it. Given an integer k and a graph G, the graph bandwidth problem asks for a function f mapping the vertices of G to the natural numbers, such that if (u, v) is an edge of G then $|f(u)-f(v)| < k$. In other words, the nodes are to be laid out on a line such that the distance between any two connected nodes is less than k. This problem is NP-complete, even if G is restricted to be a tree. Given an instance of the bandwidth problem, a cutbus can be constructed in which k segments of length k extend to the right from each node site. A graph G will be embeddable in the cutbus precisely when the bandwidth of G is less than k, and the layout in the cutbus will provide the function f.

If k is fixed in advance, however, so that the problem is to determine whether a graph G has bandwidth bounded by some constant, the bandwidth problem is polynomial [69]. Similarly, the problem of laying out a graph in a cutbus of fixed *depth* is polynomial, where the depth of a cutbus is the number of segments that can be connected to any node. The best known algorithm for layout in a cutbus is a dynamic programming algorithm that requires time that is exponential in the depth of the cutbus. The node sites of the cutbus are scanned from left to right, while a set of partial layouts is maintained. Partial layouts in node sites 1 through i are equivalent if each segment crossing the boundary between i and $i+1$ receives the same edge of the tree, with the same node on the left side

of the boundary. If the depth of the cutbus is k, and G is a tree with n nodes, then there are at most $(2n)^k$ equivalence classes of partial layouts at boundary i. The equivalence classes at boundary $i+1$ can be determined by looking at each of the classes at boundary i once for each remaining node, so that only $O(n^{k+1})$ operations are required for each node site. This algorithm thus takes $O(n^{k+2})$ time to lay out the entire tree. Note that this algorithm also works in the two-color case; all that changes is the test for making sure that updated partial layouts are legal. Layout in a cutbus of fixed depth thus seems to be feasible, either with or without reserved node sites.

3.3. A Prototype Laser-Programmable Recognizer

To demonstrate the feasibility of building and configuring a programmable layout for recognition of regular languages, a prototype laser-programmable chip called E.T.[2] was designed during the summer and fall of 1982. The chips were designed with scalable static ratio logic [57] and fabricated by MOSIS [17] using an NMOS process with four micron channels. Five chips were configured and tested during January of 1983. One of these chips was completely operational after configuration; the other four were partly operational. This experience shows that compact programmable layouts for recognizers can be constructed.

E.T. is configured for an expression by cutting metal lines that carry signals. Originally, chips are fabricated with all possible connections already made; everything is shorted together. The regular expression compiler that is used to configure the chip chooses which lines to cut and which ones to leave intact. Those lines that are to be cut are then melted with pulses from a laser. Using this type of laser configuration, the expression to be recognized by a chip can be chosen long after the chip is fabricated and bonded.

Metal lines are cut at specific locations called *programming points*. Each programming point on E.T. consists of one or more metal lines that may be cut, with a window in the overglass above them. Metal lines are 8 microns wide, with lines that are independently cuttable spaced 16 microns between edges. The overglass window covers a 16 micron length of the line, extends 10 microns past the edge of the line, and is spaced 10 microns from other features.

E.T. contains only comparator cells. A cell, shown in Figure 3-11, is about 1 mm high and 200 microns wide. It compares characters that are 4 bits wide, where each bit may be programmed to be 0, 1, or *a* (a don't-care value that matches anything). The cell includes a disable signal, DIS, as

[2] Expression Testing chip

described in Section 2.2. Two ports are at the top of the cell: a left PRA port on the left and a right PRA port on the right.[3] Within each port, the order of signals from left to right is: ENB, DIS, RES, C0, C1, C2, C3.

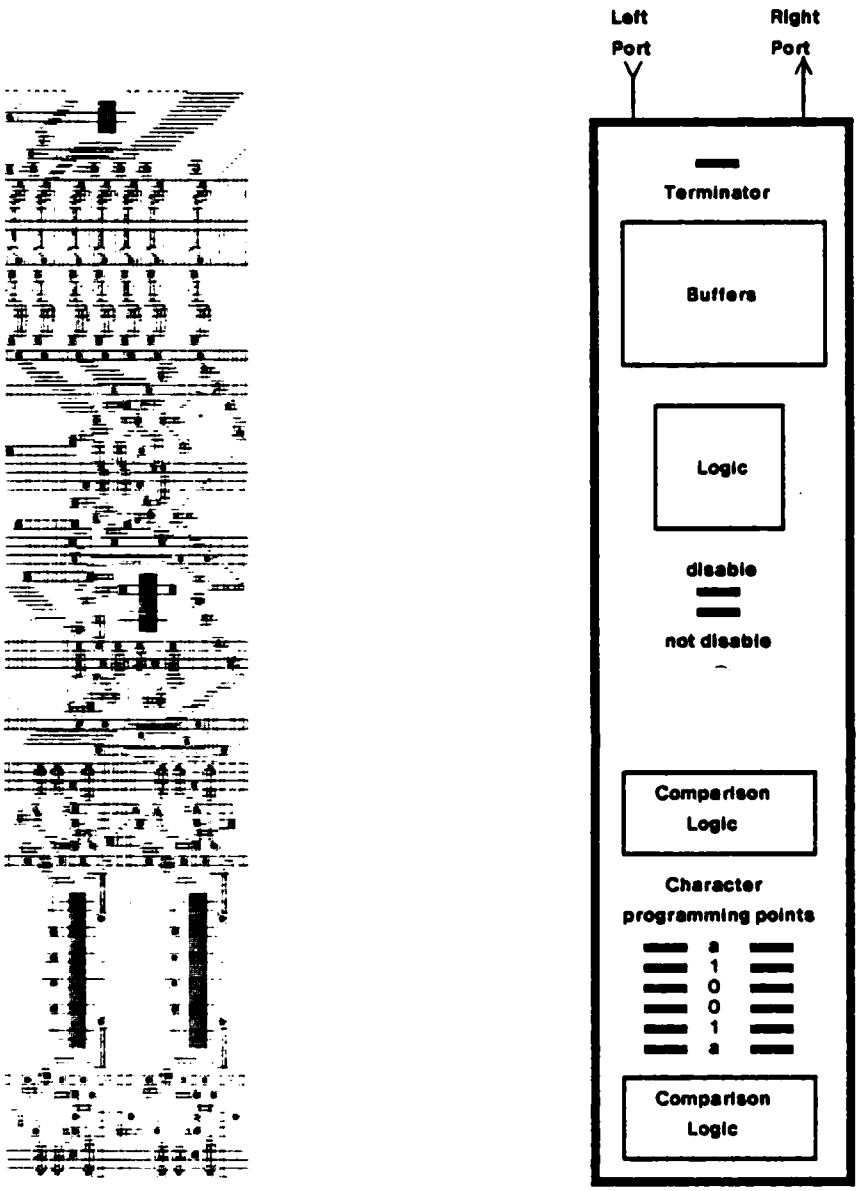


Figure 3-11: The prototype comparator cell

The cell contains 6 programming points that allow the character to be chosen, and allow the DIS and ENB outputs to interact correctly with RES. The locations and functions of the 6 programming points are:

[3] In a left PRA port, RES is an input and all other signals are outputs: in a right PRA port, RES is the only output.

- Near the top of the cell is a single cuttable wire that should be left intact iff the cell is to be terminated (*i.e.* if it's a leftmost cell).

- Near the middle of the cell is a programming point containing two wires, one of which should remain intact. The top wire should stay to make DIS disable RES, and the bottom should stay to make DIS have no effect.

- At the bottom of the cell are four programming points containing three wires each. These program the bits of the character. One wire from each point should remain intact to program 0, 1, or *a* (the don't-care). The bottom two points are mirror images of the top two, so the wire assignments are reversed. On the top points, the top wire should remain to program *a*, the middle to program 1, and the bottom to program 0. On the bottom points, the bottom remains to program *a*, the middle to program 1, and the top to program 0. The four points are arranged in a rectangle with C0 in the lower left, C1 in the upper left, C2 in the upper right, and C3 in the lower right.

Just below the topmost programming point is a set of output buffers for the two ports of the cell. These buffers are designed to drive the tree edges that interconnect the cells at high speeds. Simulation using SPICE [25] shows that a buffer can drive a single gate input through a typical RC load (4 mm of polysilicon followed by a pass transistor) at a rate of 50 Megahertz.

The delay elements within the cell are static shift register stages controlled by a two-phase non-overlapping clock. When phase 1 (φ_1) is high, the registers self-refresh and output. When phase 2 (φ_2) is high, the output is disconnected from the input and input is enabled. The output remains stable in φ_2 for as long as the gates hold their charge (over a millisecond). Thus a beat consists of the following steps.

1. Begin setting up the inputs to the cell and lower φ_1.

2. Raise φ_2 and finish setting up the inputs to the cell (making at most one transition on each input).

3. Lower φ_2.

4. Raise φ_1 and hold it high until the next beat. Cell outputs for the next beat become valid during φ_1.

This clocking scheme is compatible with the clocked OR gate shown in Figure 2-9. The output of the clocked OR gate may change from 0 to 1 during the early part of φ_2, so that φ_2 should remain high long enough to allow the E. T. comparator to accept the final value.

E.T. contains two structures made up of comparator cells: a single pre-configured comparator cell, and a configurable array of four cells. The pre-configured cell is included for two reasons: so that the

circuitry of the comparator can be tested without the additional task of laser programming, and so that any die can be tested before programming it to make sure the circuits work on that particular die. The cell recognizes the character <101*a*>, that is, bit C0 is 1, bit C1 is 0, bit C2 is 1, and bit C3 is a don't-care. The cell is not terminated, and DIS doesn't clear the RES input. Thus, DIS and ENB are merely single-stage shift registers, and

$$RES_{out}(t) = [RES_{in}(t-1) \text{ AND } (CHR(t-1)=<101a>)].$$

All inputs and outputs of the cell are connected to bonding pads.

The configurable array contains four comparator cells connected to a switching array that is eight ports long and two channels high. A channel in the array contains seven wires, corresponding to the seven wires in a port. From top to bottom, the wires in a channel are: C3, C2, C1, C0, RES, DIS, ENB. At every intersection between a port and a channel is a programming point containing 14 wires, as shown in Figure 3-12. The channels in Figure 3-12 run horizontally in metal, while the ports run vertically in polysilicon in the left half of the figure. The long, narrow rectangle in the right half of Figure 3-12 is the overglass window. The channel can be disconnected from the wire by cutting seven of the wires in the programming point (wires 2, 3, 6, 7, 10, 11, and 14, counting from the top). By cutting the other seven (1, 4, 5, 8, 9, 12, and 13), the channel can be split to the left of the port. Section 3.2.2 shows how to use fusible links of this type to configure a recognizer.

Figure 3-13 is a checkplot of the entire chip, which is about 2.8 mm on a side. The configurable array is the large structure to the left of center and the test cell is the smaller structure to the right of center. Bonding pads are arrayed around the edge of the chip and wired to the two internal structures. The two bars near the upper right corner are marks distinguishing this chip from earlier versions.

In addition to V_{dd}, ground, and the two clock phases, there are several signals in the two structures that are connected to bonding pads. All inputs and outputs of the test cell are so connected. In addition, the top channel of the array is connected to two sets of bonding pads. The left end of the channel is connected to pads for a left PRA port, and the right end of the channel is connected to pads for a right PRA port. When E.T. is configured, these pads may be connected to the ports of any of the cells. An input pad is connected to diodes along the top edge of the array, so that it may be precharged during φ_1. Starting with the leftmost pad on the top edge of the chip, and going clockwise, the bonding pads are:

- precharge
- right port of configurable array (7 pads): c3 in, c2 in, c1 in, c0 in, RES out, DIS in, ENB in

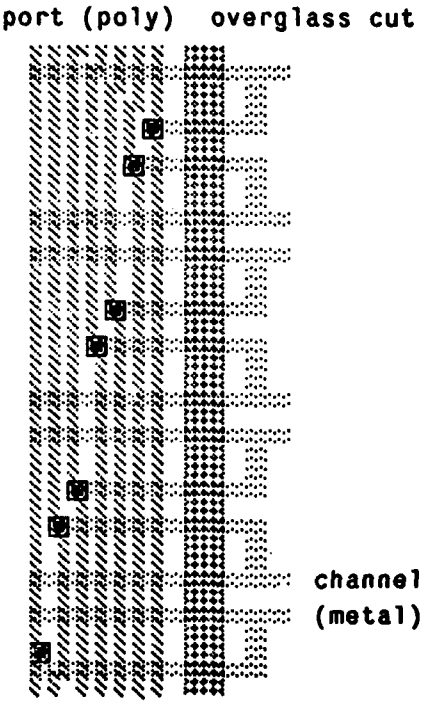


Figure 3-12: A programming point in the connection network

- ground
- V_{dd}
- left port of <101*a*> cell (7 pads): ENB out, DIS out, RES in, c0 out, c1 out, c2 out, c3 out
- right port of <101*a*> cell (7 pads): ENB in, DIS in, RES out, c0 in, c1 in, c2 in, c3 in
- φ_1
- φ_2
- left port of configurable array (7 pads): ENB out, DIS out, RES in, c0 out, c1 out, c2 out, c3 out

Chips were fabricated and bonded by MOSIS in late 1982 and the test cell on each chip was tested at CMU. All eight packages returned by MOSIS were operational at a clock rate of 100 nanoseconds per beat. Five of the chips were therefore laser programmed at the RVLSI facility at MIT Lincoln Laboratory during January of 1983. The laser used for programming is focussed to a four micron spot size through a microscope objective that permits simultaneous inspection. The chip is mounted

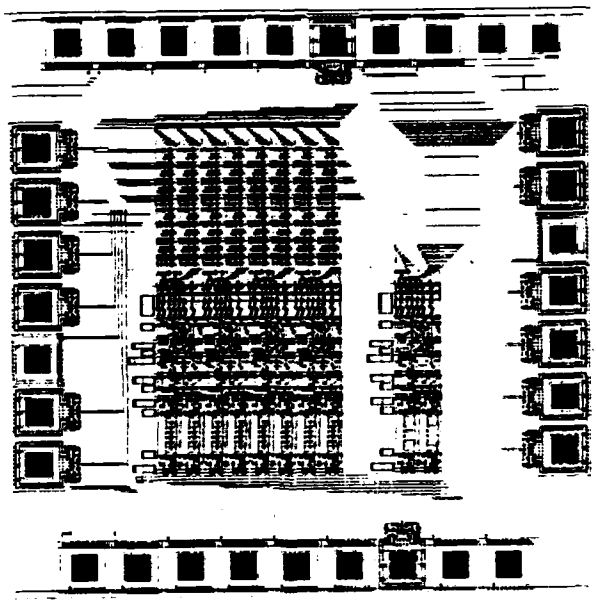


Figure 3-13: The laser-programmable chip

on a computer-controlled motorized stage, so that a pattern of laser cuts can be created and checked before the chip is actually configured.

Although other laser-programmable chips have been constructed [48, 62, 71, 74], this was the first such chip fabricated by MOSIS. Calibration of the laser was thus the first step in programming, since small variations in laser parameters can cause large differences in effects. Ideally, the metal should be cut reliably without damaging the oxide beneath.

Previous experiments with chips produced at Lincoln Laboratory provided a candidate cutting technique, in which each wire within a programming point was cut using 13 laser pulses arranged in the pattern shown in Figure 3-14. The pulses lasted about a millisecond at a power level of 2 watts. Since E.T. did not include separate calibration structures, we tried this technique on one of the lines in the switching array that connects an input pad to an output pad. We first attempted to isolate this line from the rest of the array by cutting eight lines. Since the output pad could be controlled by the input pad after cutting, but not before cutting, we deemed this a success. We confirmed the experiment by cutting the line, and showing that the output could no longer be controlled.

The five chips were configured using two different expressions. Three of the chips were configured

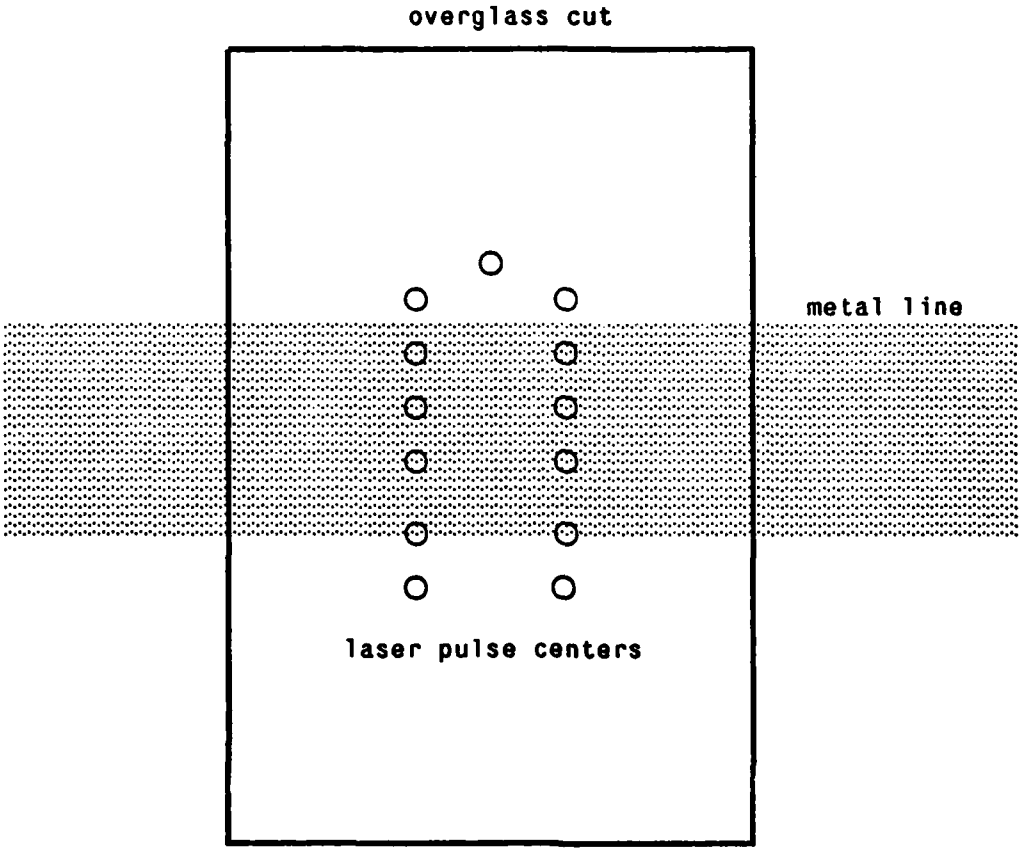


Figure 3-14: The pattern of laser pulses

for the three-character pattern PRA[4] and the other two were configured to be the same as the test cell (<101a>). Figure 3-15 shows the sites that were cut for the pattern PRA and Figure 3-16 shows the sites for <101a>. Configuring each chip took about 18 minutes.

After configuration the chips were tested for function. To ease testing, five of the outputs (C0, C1, C2, C3, and ENB) from the leftmost cell of the PRA chips were connected to output pads during configuration.[5] One of the <101a> chips worked completely and the other four chips were partially operational. Table 3-1 gives details of the configured packages and test results.

---

[4]Bit representations for P and R have the lower three bits of the ASCII representation in C0, C1, and C2, and the wild card a in C3. The character A contains a in all four positions.

[5]On one package (8) the output pad for C3 was disconnected during calibration of the laser.

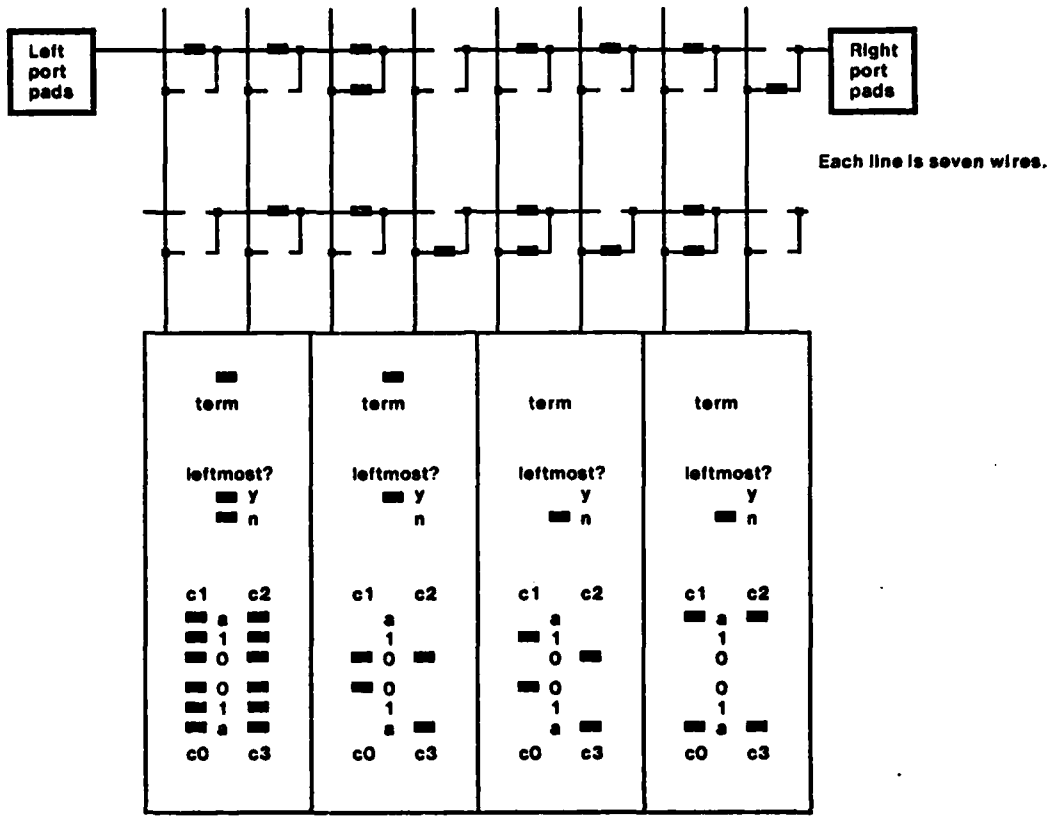


Figure 3-15: Cut points for the pattern PR*A*

Since the test cells on all chips worked, we can conclude that most of the faults in the configured chips were caused by laser cutting. Two types of faults may occur: *bridging* faults in which lines are incompletely cut and *shorting* faults in which cut lines are shorted to other parts of the chip (such as the substrate). From the test results, we can estimate the rates of occurrence for these types of faults.

Of 14 three-stage shift registers whose test results are shown in Table 3-1, 10 worked. For every working three-stage shift register, eight cuts can be shown to have no faults of either type. An additional five cuts can be shown to have no shorting faults. Assuming that fault occurrences on separate cuts are independent, the fraction of fault free cuts can be estimated as $(10/14)^{1/8}$, or 95%. Similarly, the fraction of cuts with no shorting faults may be estimated as 97%.

Electrical tests made after configuration show that shorting faults exist on all of the configured chips except for package 8. The substrate on MOSIS chips is connected to a package pin. Since all of the lines in the switching array are driven by pullup-pulldown pairs, shorts to the substrate can be

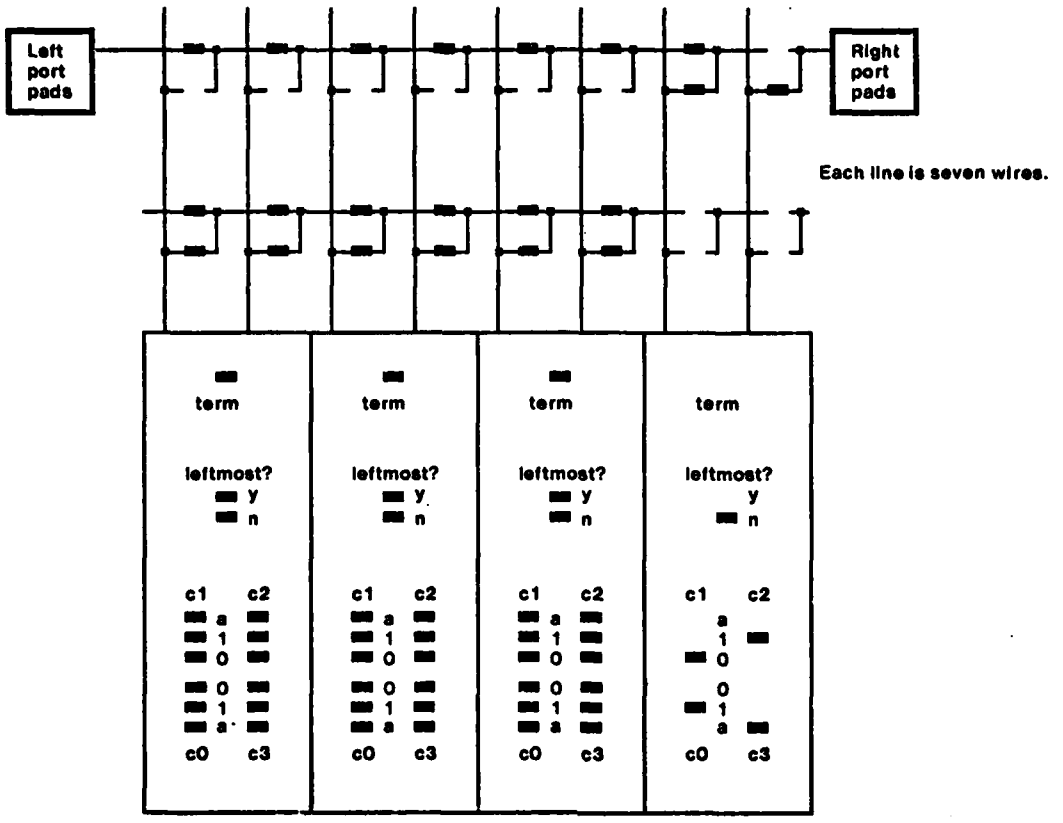


Figure 3-16: Cut points for the pattern <101*a*>

detected by applying 5 volts to V_{dd}, 0 volts to ground, and 0 volts to a resistor attached to the substrate, as shown in Figure 3-17. Any positive voltage at the substrate pin indicates a shorting fault. On all unconfigured chips the voltage at the substrate remained at 0, while four of the configured chips had substrate voltages of about 0.1 volt. This indicates shorting faults on these chips.

To further explore the causes of failure, the chips were examined using an electron microscope with an X-ray spectrometer. The working shift registers pinpointed some fault-free cuts while the faulty shift registers indicated cuts that were potentially faulty. Figure 3-18 shows a cut that is fault free. Note that the metal line is cut cleanly and that no metal has flowed into the hole in the oxide. Figure 3-19 shows a bridging fault, in which aluminum still connects the two halves of the cut. (X-ray analysis indicated that the material in the cut in Figure 3-19 contains significant amounts of aluminum.) It seems likely that Figure 3-19 is also a shorting fault. All cuts showed oxide damage, indicating that the power of the laser was too high. The bridging faults are probably due to re-joining of the metal during the multiple pulses. It seems that the pulses used in configuring E.T. applied too much energy per unit area, with insufficient uniformity.

Package	Pattern	Test Results
1	PR*A*	ENB, C0, C3 shift registers OK. C2 and RES outputs stuck low. C1 output stuck high.
2	PR*A*	C0, C1, C2 OK. C3 and ENB don't follow input. RES is correct if ENB is held high for two beats instead of one.
3	<101*a*>	C0 and ENB OK. C1, C2, C3 stuck high.
4	<101*a*>	Completely operational.
8	PR*A*	C0, C1, C2, ENB OK. C3 was cut off from output during laser setup. RES stuck low.

Table 3-1: Test Results for configured chips

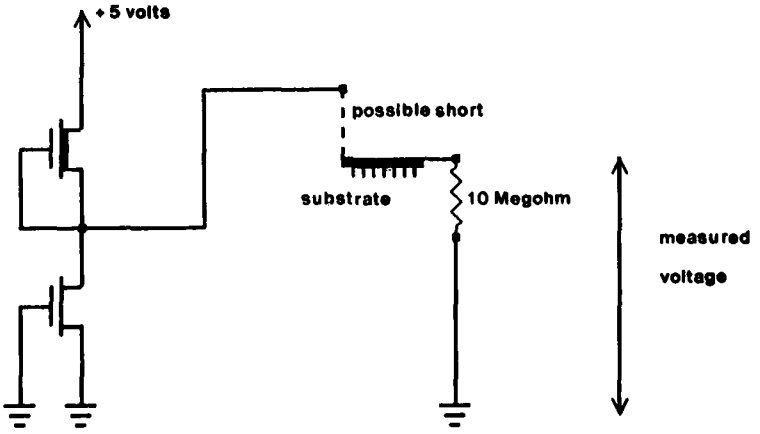


Figure 3-17: Test setup for detecting shorting faults

Based on the experience with the first prototype chip, we designed a second version of the chip. Calibration structures consisting of 15 programming points connecting two pads were included on the second version. Using these calibration structures, a new laser cutting procedure was developed, in which each programming point was cut using a single 2.8 watt 1 millisecond pulse from the laser, focussed to a 10 micron spot size. This pulse was applied through the overglass itself, rather than through the overglass window. Using this new technique, we were able to make about 300 cuts on

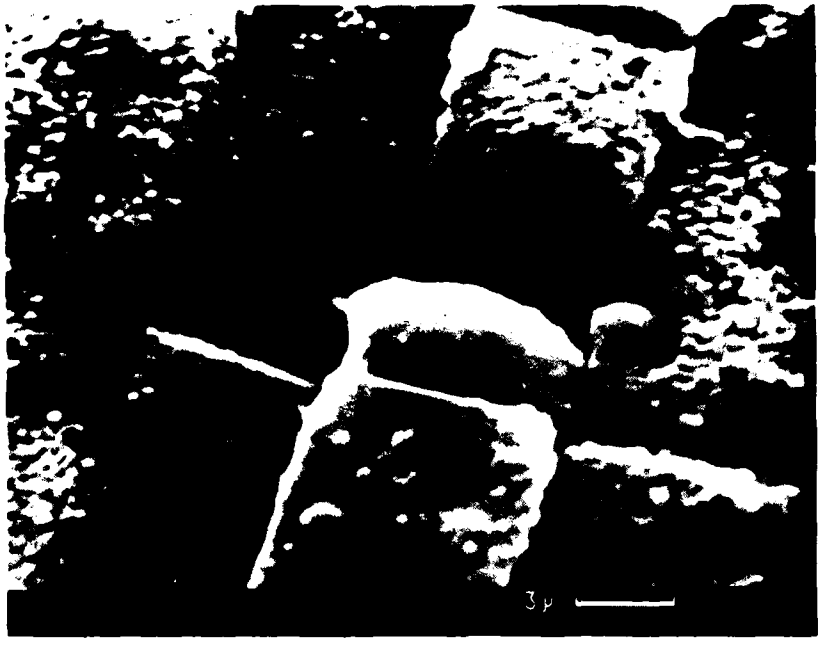


Figure 3-18: A fault-free cut

Figure 3-19: A bridging fault

two chips without a single bridging or shorting fault. The first chip was not operational after cutting, but microscopic examination revealed some incomplete polysilicon lines, probably caused by mask defects. The second of the chips was then programmed for a pattern that permitted testing everything except the defective area. All other parts of this second chip were operational.

From the experiments with these prototype chips, we can conclude that configuration after fabrication is feasible, though research is needed into configuration techniques. Experiments are required to find acceptable laser settings for MOSIS chips. NMOS chips are susceptible to shorting faults; laser pulses have been used to deliberately make connections between metal and substrate [48]. New link structures should also be evaluated. In particular, it seems that the overglass window in Figures 3-12 and 3-14 should be eliminated. The etching step used in cutting the windows may also etch the field oxide that insulates the metal from the substrate, making shorting faults more likely. The uncertainties of laser cutting through overglass seem less harmful than this uncontrolled etch. A set of test chips should be fabricated and programmed to investigate link structures and laser parameters.

As in the second version of E.T., calibration structures for the configuration process should be included on future laser-programmable chips. These structures could consist of cuttable metal lines between probe points. The lines could be cut with varying power on the laser and the points could then be probed for shorts and bridges to find an appropriate power setting. With calibration experiments and the inclusion of these structures, configurable layouts could be attractive in many applications. Using cells similar to the E.T. comparator, single chips could be built that could be configured for regular expressions of length 30 to 70. Programmable layouts are therefore worthwhile in conjunction with specialized silicon compilers.

Chapter 4
A Comparative Survey of Recognizers

Chapters 2 and 3 have presented a scheme for compiling a regular language into a recognizer. This is not the only such scheme. A surprising number of recognizers can be constructed for any regular language, all seemingly different. Depending upon the individual application, one or another of these schemes may be smaller or faster. Which of the recognizer schemes is superior in an application depends upon the language to be recognized, the tasks to be performed other than language recognition, and other details of the application.

This chapter surveys several types of recognizers for regular languages that may be suitable for VLSI implementation. A specialized silicon compiler might choose among the schemes presented here, basing its choice on the application area, the language to be recognized, and any speed or area restrictions in the target hardware. The aim of this chapter is to provide a guide for selecting recognition algorithms. Several schemes for constructing recognizers will be described briefly, then the recognizers will be compared on the basis of speed, space, and extensibility. The different design approaches will be illustrated using one example: the language L given by the regular expression $1(1 + 0^+)1$. Thus, $L=\{111, 101, 1001, 10001, \ldots\}$.

4.1. Automata-Based Recognizers

Any regular language can be recognized by a finite-state automaton. A finite-state automaton has a finite set of states, with one called the *start* state and a subset of states called *final* states. The automaton is initially in the start state. Each input character may trigger one or more *state transitions*, putting the automaton in a new state that depends upon both the state beforehand and the input character. An input string is recognized if and only if the automaton is in a final state after all characters have been input. The language recognizers discussed in this section are direct realizations of finite-state automata for the regular language.

4.1.1. Minimum-State Deterministic Automaton

A deterministic automaton is one in which precisely one state transition is triggered by any input. Any regular language has a unique deterministic finite-state automaton with a minimal number of states, which can be constructed in polynomial time from any larger automaton [38, 59]. The minimum-state deterministic automaton for L is shown in Figure 4-1. Each of the circles in the figure represents a state, with a double circle for the final state; the labeled arrows represent state transitions. The start state is the state with an unlabeled arrow pointing into it.

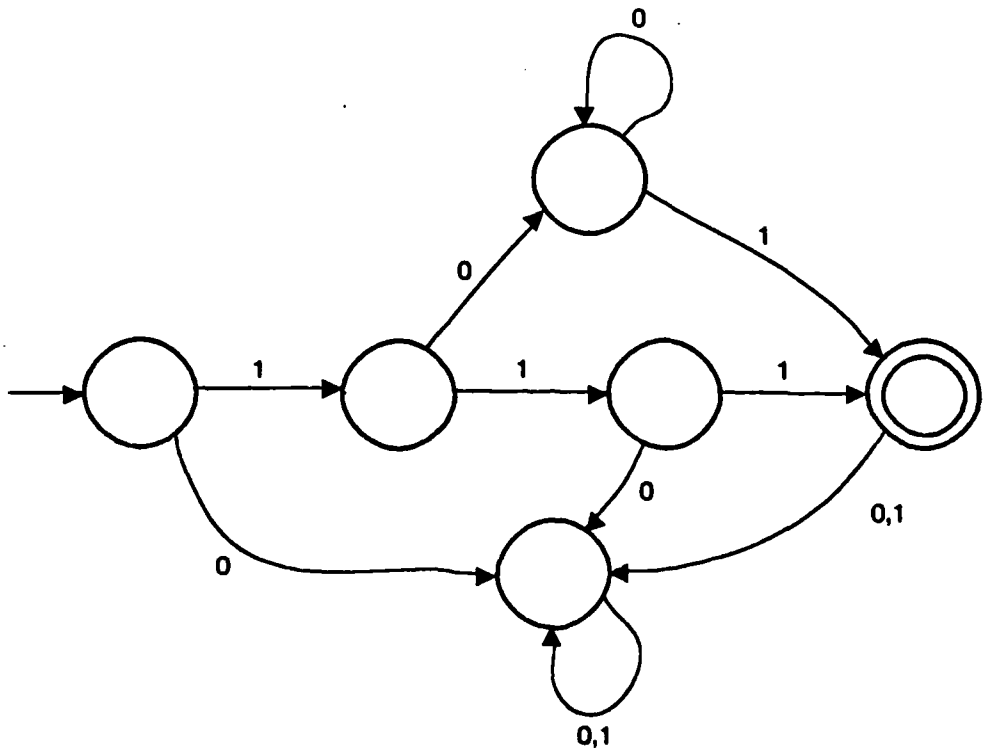


Figure 4-1: Deterministic finite-state automaton for $1(1 + 0^+)1$

An n-state deterministic automaton can be realized using several methods. Classically, a set of $\lceil \lg n \rceil$ flip-flops is used to record the state, and transitions are implemented using combinational logic. The problem of assigning states of the flip-flops to states of the automaton so that the combinational logic is minimized has been extensively studied [44]. In modern practice, a microprocessor is often used, with a state table containing $O(n \lg n)$ bits held in main memory and characters input using the i/o system. Microprocessor realization allows easy programming of the state transition function.

4.1.2. Non-Deterministic Automata

Non-deterministic finite-state automata are often considerably simpler than the corresponding deterministic automata, with fewer states and fewer transitions. A non-deterministic automaton can be thought of as being in several states at the same time, or in no state at all. Figure 4-2 shows a non-deterministic automaton for L. If a 1 is input while the automaton is in the start state, it will go into two new states. On the other hand, if a 0 is input the automaton goes into no state.

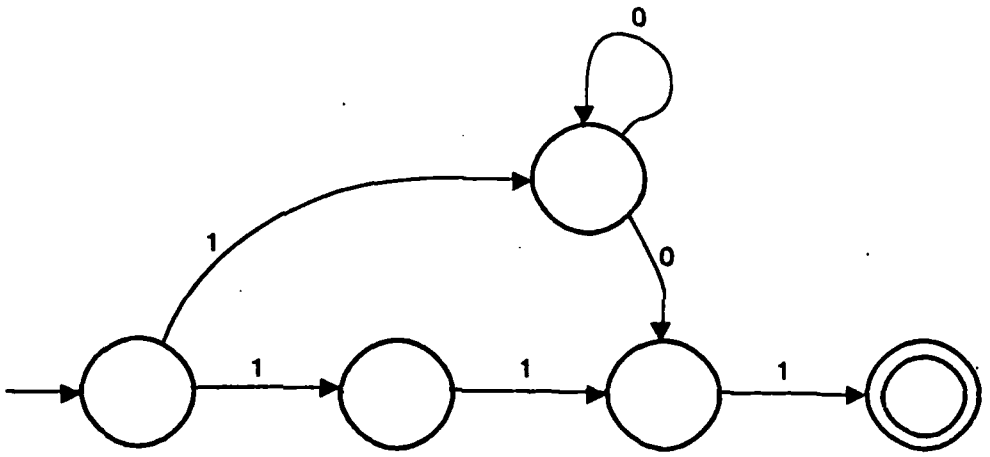


Figure 4-2: Non-deterministic finite-state automaton for $1(1 + 0^+)1$

The problems of realizing a non-deterministic automaton are similar to those for deterministic automata. An encoding for the state must be chosen, along with an implementation of the state transition logic. Two direct realizations of non-deterministic automata have been reported that solve these problems in different ways.

Floyd and Ullman [28] have presented a realization using a *state register*, in which each state in the automaton is assigned a bit. All state transitions are computed in parallel using combinational logic. They originally proposed using a single PLA to update the state register, but further work by Trickey [75] has shown that using several smaller PLA's can offer a significant area improvement.

Haskin [34] has presented a realization using an ensemble of deterministic machines. Each machine uses a random-access memory to hold both its state and transition function, and uses a special-purpose processor to compute the new state from the old. Whenever more than one state is entered, additional machines are started in the extra states. Thus, if the automaton is in k states, k machines from the ensemble will examine the next input. To avoid dynamic activation and

passivation of machines, the states of the non-deterministic automaton are divided into *compatible sets*, such that the automaton is never simultaneously in two states from a compatible set. Machines are then pre-allocated, one for each compatible set.

4.2. Expression-Based Recognizers

These recognizers are constructed by selecting a regular expression for the language. The recognizer can be derived automatically from the expression. The compiler described in Chapter 2 produces circuits of this type. In addition, one other expression-based recognizer scheme has been reported independently by several authors.

4.2.1. Systolic Recognizer

A complete description of this recognizer and its layouts can be found in Chapters 2 and 3, so only a sketchy description is given here. A set of primitive cells is designed, one cell for each character that may appear in an expression. A syntax-directed technique is used to interconnect these cells into a recognizer. The circuits formed in this way are ternary trees, and so can be laid out using any of several well-known techniques. A systolic recognizer circuit for L, using a single cell for the Kleene + operator, is shown in Figure 4-3.

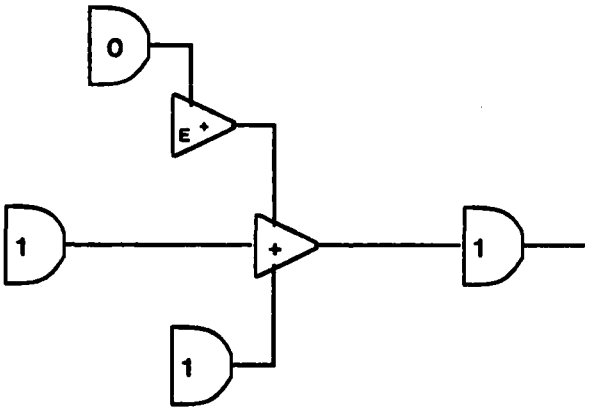


Figure 4-3: Systolic recognizer for $1(1 + 0^+)1$

4.2.2. Expression-Tree Recognizers

Several researchers [28, 60] have independently discovered a recognizer scheme based on the expression tree of a regular expression. As in the systolic recognizer, a set of primitive cells is interconnected to form the recognizer. In this case, however, there is one primitive cell for each operator that may appear in an expression, including concatenation. In addition, a separate comparator cell is used for every character in the expression. A recognizer circuit formed from these cells has the same form as the expression tree of the regular expression.

Figure 4-4 shows the comparator for the expression tree recognizer. On each beat, a character is input at the same time as the ENB signal. The RES signal is set to true for the following beat if and only if ENB is true and the text character matches the pattern character.

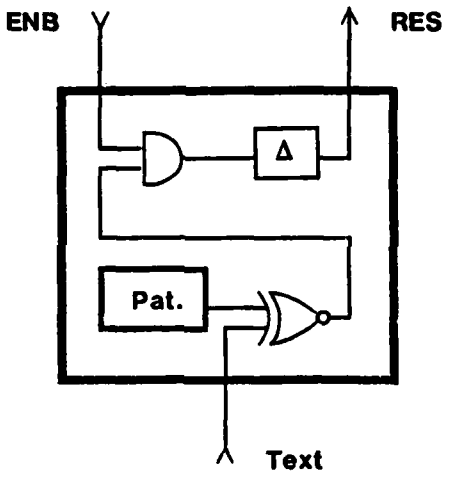


Figure 4-4: Comparator for expression-tree based recognizer

The three operator cells for the expression tree recognizer are shown in Figures 4-5, 4-6, and 4-7. These combine ENB and RES signals from their operands to produce signals for a larger expression. For example, to build a recognizer for AB, a comparator for A is connected to the left port of the concatenation cell, and a comparator for B is connected to the right port. A recognizer constructed using these cells outputs RES on beat i if and only if some string in the language of the recognizer is input on beats $i-n$ through $i-1$ and ENB_{i-n} is true. As in the systolic recognizer, a clocked OR gate, similar to the one in Figure 2-9, must be used in the Kleene closure cell to prevent latch-up.

Figure 4-8 shows the recognizer for L. The set of primitive cells for expression tree recognizers can

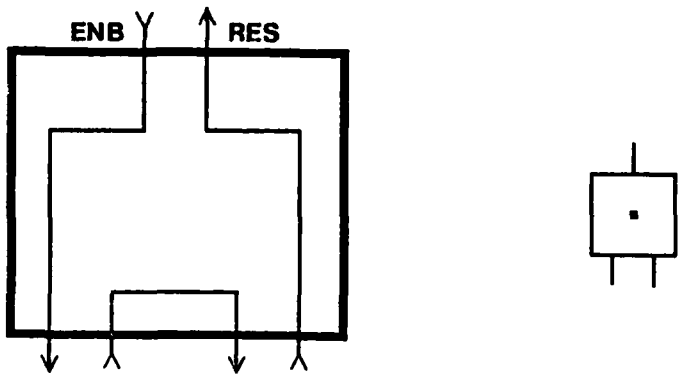


Figure 4-5: Concatenation cell

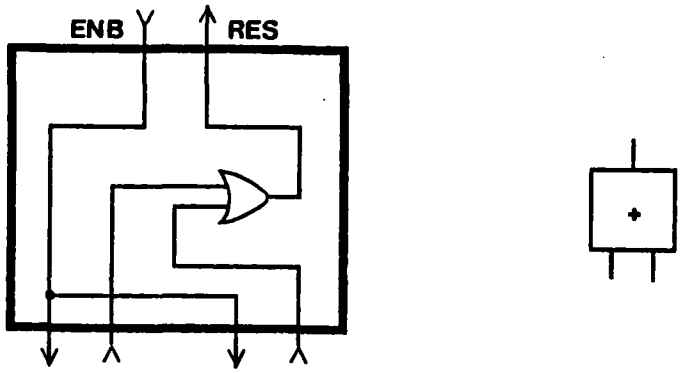


Figure 4-6: Union cell

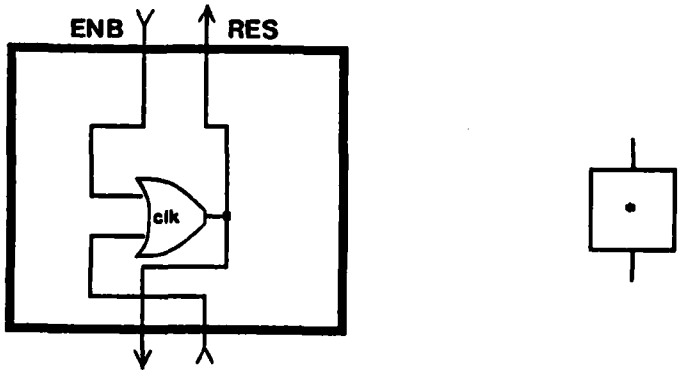


Figure 4-7: Kleene closure cell

be extended for additional operations, in the same manner as the cells for systolic recognizers. The circuit in Figure 4-8 uses an extended cell for the Kleene + operator.

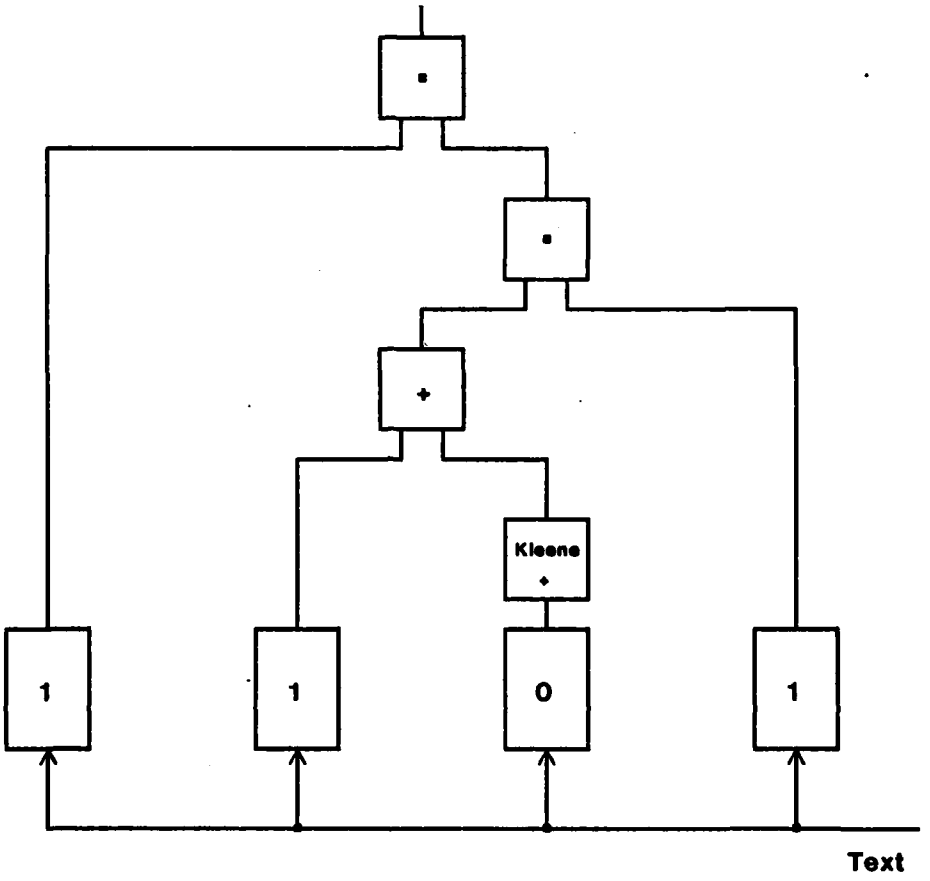


Figure 4-8: Expression tree recognizer for $1(1 + 0^{+})1$

Although the two expression-based recognizers are superficially similar, several differences should be noted. One important difference is that the non-systolic recognizer described in this section uses more broadcast than the systolic recognizer of Chapter 2. Figure 4-8 shows that the text characters in the non-systolic recognizer are broadcast to all comparators. For large recognizers of this type, the input characters must be distributed to the whole chip on every beat, which may cause the non-systolic recognizers to run slowly. Another difference is that the cells in the non-systolic recognizer are simpler. Each character cell has only one delay element in it, rather than the three found in the systolic comparator cell. This may allow non-systolic recognizers to have smaller area. Selection between these methods depends crucially on the details of the implementation technology.

4.3. Other Recognizers

Several recognizers have been proposed that are not based directly upon either expressions or automata.

4.3.1. Grammar-Based Recognizer

Chu and Fu [15] have shown how to construct a recognizer for any context-free language, based on a grammar for the language. Since every regular language is context-free, a simplification of this construction can be used for regular languages. At every time step, this simplified recognizer inputs a character c from the input string, along with a set S_{in} of non-terminal symbols and computes a new set S_{out} of non-terminals. The computation may be represented as a product:

$$S_{out} \leftarrow S_{in} \cdot c.$$

This product is computed by examining the productions in the grammar. The non-terminal P is in S_{out} if and only if one of two conditions is met:

- there is some production $P \rightarrow Qc$, where $Q \epsilon S_{in}$ and c is the input character;
- c is the first character of the input string, and there is a production $P \rightarrow c$.

One cell of this type functions as a recognizer. On the first beat, S_{in} is set to $\varnothing$, and the first character of the input string is input to the cell. On each succeeding beat, S_{in} is set to S_{out}, and the next character is input. At the end of the input S_{out} is checked to see whether it contains the start symbol of the grammar. The string is recognized if and only if the start symbol is in S_{out} after the last character of the string has been input.

This method of computing the product requires a left-linear grammar for the language. In the original presentation, right-linear grammars were used, with the undesirable effect of requiring the input string to be in reverse order. Since any regular language has both right-linear and left-linear grammars, the two formulations are equivalent.

Since a linear grammar for the language is required, these grammar-based recognizers are equivalent to automata-based recognizers. Any left-linear grammar corresponds directly to a non-deterministic finite-state automaton with one state for each non-terminal. The automaton contains a transition on input c from the state corresponding to P to the state corresponding to Q if and only if there is a production $Q \rightarrow Pc$ in the grammar. For example, the left-linear grammar corresponding to Figure 4-2 is:

$L \rightarrow C1$
$C \rightarrow A1 \mid B0$
$B \rightarrow 1 \mid B0$
$A \rightarrow 1$

where L is the start symbol for the grammar. A similar correspondence exists for right-linear grammars. The task of designing a grammar-based cell for a regular language is thus exactly the same as designing an automata-based recognizer: the problems of state encoding and logic realization are unchanged. Any state encoding for a non-deterministic automaton corresponds directly to an encoding for the sets of non-terminals S_{in} and S_{out}. Once an encoding is chosen, the cell for a grammar based recognizer must have exactly the same logic as is required for state transitions in the non-deterministic automaton.

Chu and Fu proposed a mesh of these recognizer cells for context-free language recognition and noted that the mesh could be replaced by a linear pipeline for regular languages. A pipeline of n recognizer cells can act in parallel to recognize n input strings, each of length n, simultaneously. The S_{out} signal from one recognizer provides the S_{in} signal to its right-hand neighbor, and character i from each input string is fed to cell i. The data flow in this pipeline is identical to Kung and Leiserson's systolic matrix-vector multiplier using inner-product cells [51]. Figure 4-9 shows a pipeline of length four. The strings in Figure 4-9 are $\langle a_1a_2a_3a_4\rangle$, $\langle b_1b_2b_3b_4\rangle$, and so on, so that a_3 is the third character of the first string. At beat 1, input character a_1 is sent to cell 1. At beat 2, cell 2 receives cell 1's output, together with input character a_2, and computes its own output. After n beats, the start symbol is checked for membership in S_{out} of cell n. Thus, after a latency of n beats, one match result for a string of length n emerges from the pipeline on each beat.

Use of this pipeline in a regular language recognizer does not seem to be worthwhile, though the mesh may be needed for more general context-free languages. A set of n matches can proceed simultaneously by using one of the n cells for each input string, feeding its S_{out} back into S_{in}. This eliminates the dependence of string length on the number of cells, and permits more flexibility in input timing. Since the cells of a grammar-based recognizer can be designed as automata-based recognizers, and since pipelining them confers no advantage, grammar-based or pipelined recognizers will not be considered further. No specialized silicon compiler that is restricted to regular language recognition should include them as a design alternative.

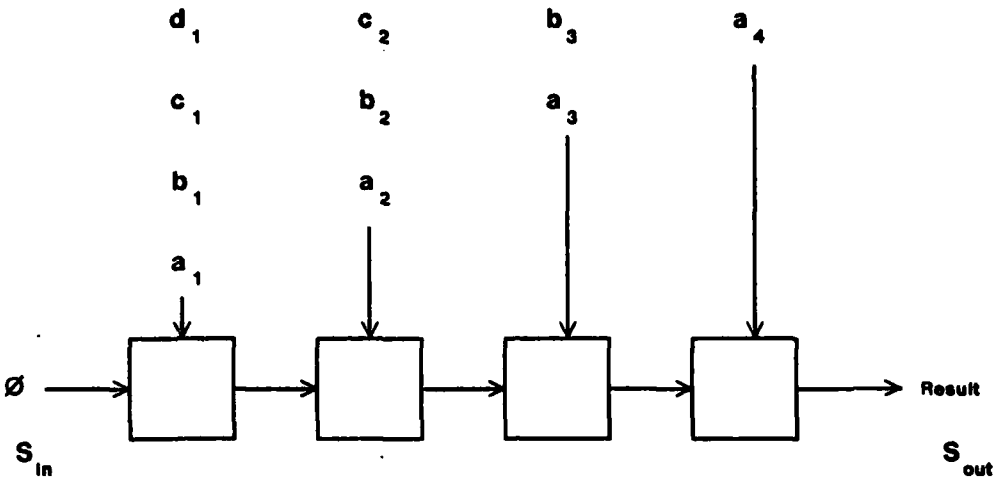


Figure 4-9: Linear pipeline for regular language recognition

4.3.2. Monoid Composition

An algebraic object called the *syntactic monoid* is associated with every set of strings. The syntactic monoid consists of a set of elements with a binary operation (called *composition*) and an identity element. Computations within the syntactic monoid of a regular language can be used to determine whether a string is in that language. This section defines the syntactic monoid of a language and shows how to construct a recognizer based on monoid composition.

Any set of strings $S \subseteq \Sigma^*$ over an alphabet Σ induces a natural equivalence on Σ^*, the set of all strings. Strings s_1 and s_2 are equivalent in the relation induced by S whenever, for all strings α and β, $\alpha s_1 \beta \in S$ iff $\alpha s_2 \beta \in S$. The equivalence classes under this relation form the syntactic monoid, which is written syn(S). If the equivalence classes of strings ρ and σ are denoted $[\rho]$ and $[\sigma]$, then composition in syn(S) is defined by:

$$[\rho] \cdot [\sigma] \triangleq [\rho\sigma].$$

Using this composition rule, the syntactic monoid of a set of strings has identity element $[\varepsilon]$ (the equivalence class of the null string).

The syntactic monoid of a regular language is finite and can be described using a transition diagram similar to a finite-state automaton [63]. Figure 4-10 shows the syntactic monoid of the language L. The circles represent monoid members and the labeled arrows are used in computing monoid products. To compute the product $[\rho] \cdot [\sigma]$ in the monoid, start at the circle labeled ρ, and follow the arrows with labels that make up σ. For example, $[10] \cdot [01] = [101]$.

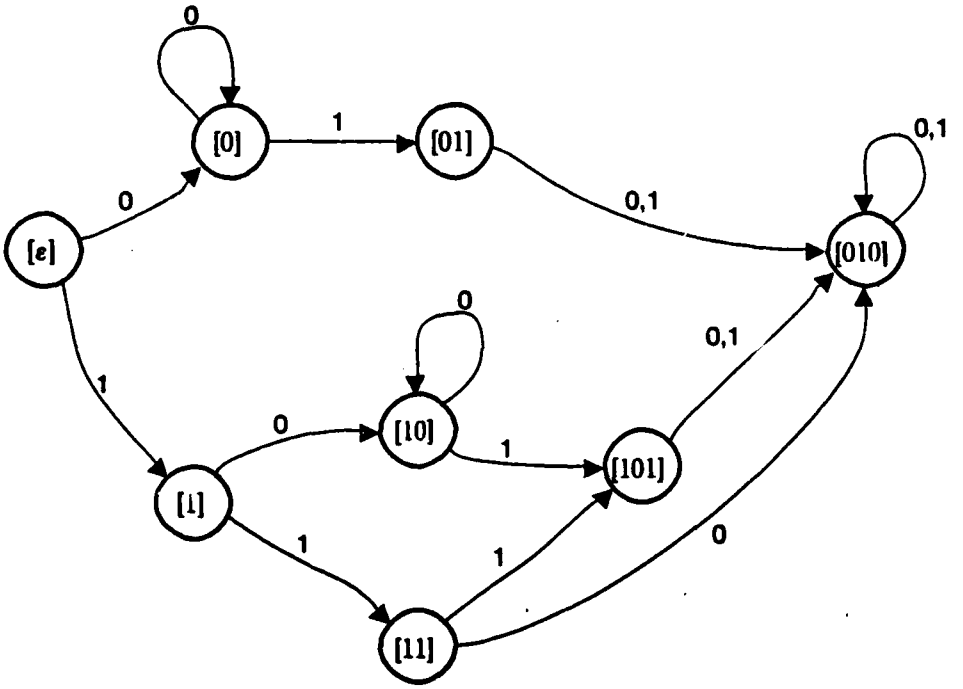


Figure 4-10: Syntactic monoid for $1(1 + 0^+)1$

Another representation of the syntactic monoid of a regular language is as a monoid of square boolean matrices. If the language L has an automaton (deterministic or not) with s states, syn(L) is isomorphic to a monoid of $s \times s$ boolean matrices under the product operation. Each element of the monoid may be thought of as a mapping taking each state of the automaton to a set of states. Let the states of the machine be $S_1 \ldots S_s$. Then the (i, j)'th element of a monoid element is 1 iff state S_i is in the image of state S_j under the mapping. Since [ε] takes each state to itself, for example, [ε] corresponds to the identity matrix. To take another example, if $c \in \Sigma$ then the matrix corresponding to [c] has a 1 in position (i, j) if and only if the automaton contains a transition labeled c from state j to state i.

The syntactic monoid of a set S can be used to determine membership of a string in S. Since no string that is in S is equivalent to any string not in S, some set C of equivalence classes contains all members of S. Furthermore, if a string is a member of a class in C, then the string is in S. Thus, by computing the monoid product of a string σ, and testing that for membership in the set C of monoid elements, membership of σ in S can be tested.

This membership test is conceptually simple in the case of regular languages, which have finite syntactic monoids. The set C of equivalence classes of members of the language must also be finite, so the monoid product of a string to be recognized must be tested for membership in a finite set. For example, in Figure 4-10, strings in L are all in the equivalence class [101], so that $C = \{[101]\}$. The string 1001 can be tested for membership in L by computing the products $[1]\cdot[0]\cdot[0]\cdot[1] = [10]\cdot[01] = [101]$. This computation shows that $1001 \in L$.

As Culik and Jürgensen have pointed out [20], since composition in the syntactic monoid is associative the test for membership can proceed in parallel using a fan-in tree. To test membership of a string $s_1 s_2 \ldots s_n$ of length n, the products $[s_1]\cdot[s_2]$, $[s_3]\cdot[s_4]$, $\ldots$ $[s_{n-1}]\cdot[s_n]$ can be computed first, followed by the products $[s_1 s_2]\cdot[s_3 s_4]$, $[s_5 s_6]\cdot[s_7 s_8]$ $\ldots$, and so forth, until the product of the whole string is computed. If as many products as possible are computed in parallel, with each product computed in time $T(L)$, only $O(T(L)\cdot \log n)$ time is required to test a string of length n for membership in the language L.

4.4. Comparison of Recognizers

Although all of the construction methods presented in this chapter produce correct recognizers, they differ in several aspects that may be relevant in applications. Some of the methods produce small recognizers, some produce fast ones, and some make extending the functions of recognizers easy. Depending on the language to be recognized and on the application, a specialized silicon compiler might choose one or another of the construction methods. Selection of one scheme over another depends greatly on the technology used to implement recognizers, but some general guidelines can be given. This section presents some of the considerations that should be taken into account when choosing a type of recognizer to build.

In VLSI design, minimization of area is often an overriding concern. The area of a recognizer depends on both the language to be recognized and the type of the recognizer. The area of an automaton-based recognizer, for example, depends on the number of states in the automaton chosen for the language. The area of an expression-based recognizer depends on the length of the chosen regular expression. Two theorems will show that for some languages, automata-based layouts have minimal area, while expression-based layouts have minimal area for some other languages. If the area of a recognizer must be minimized, then, a specialized silicon compiler should be able to choose between at least these two types.

Theorem 4-1 shows that no matter what kind of recognizer is used, some languages with s-state

deterministic automata require $\Omega(s \log s)$ area for layout of a recognizer circuit. This bound is tight, since it can be achieved by using a small processor together with a state table of $O(s \log s)$ bits. For each combination of state and input character, the table contains the next state encoded in $\lceil \lg s \rceil$ bits. The processor uses this encoding as an index into the table on each transition. Theorem 4-1 shows that the area of this automata-based recognizer is asymptotically optimal for some languages.

Theorem 4-1: For any algorithm for layout of recognizers, and for any choice of s, there is some language with s states in its minimum deterministic automaton whose recognizer layout takes $\Omega(s \log s)$ area.

Proof: The proof of this theorem depends upon finding a set of s^s different languages with s-state automata over the alphabet $\{0, 1\}$. Constructing a recognizer for an arbitrary language from this set within some area is equivalent to writing $s \lg s$ bits in that area. The bits can be read by presenting strings to the recognizer, thereby determining which language is recognized. Any method for laying out recognizers must therefore use $\Omega(s \log s)$ area for at least one of the languages.

Consider the following family F_s of s-state machines with input alphabet $\{0, 1\}$. Let the states of a machine in F_s be numbered $0, 1, 2, \ldots s-1$, with state 0 being both the start and final state. F_s consists of all machines over $\{0, 1\}$ such that for every state i, an input of 0 causes a transition from state i to state $(i+1) \bmod s$. Any string of zeroes whose length is a multiple of s is accepted by every member of F_s.

To complete the proof, we show that F_s contains s^s machines, and that each of the machines in F_s represents a different language. This will show that a method for laying out recognizers for machines in F_s must lay out one of s^s different circuits, and so requires $\Omega(s \log s)$ area.

To see that F_s contains s^s machines, consider the effect of an input of 1 in each possible state of a machine. Each machine in F_s can be represented by a vector of length s, in which component i is the name of the state that follows state i on input 1. There are s possibilities for each component, and s components, so there are s^s different machines.

To show that different machines in F_s accept different languages, we choose a pair of machines and exhibit a string that is in the language of one of the machines, but not the other. Let P and Q be different machines. Then there is some state i such that input 1 causes transitions to different states in the two machines. Suppose P has a transition on 1 from state i to state p and Q does not; then P accepts the string $0^i 1 0^{s-p}$ while Q does not. The machines therefore correspond to distinct languages, so that there are s^s languages in F_s, one for each mapping of states to states.

The number of bits required to specify a particular language in F_s is $\lg (s^s) = s \lg s$. Any recognizer layout method using less than $\Omega(s \log s)$ area for every language in F_s could be used to store n bits in less than $\Omega(n)$ area. Thus, no such method exists.

□

Theorem 4-1 shows that no layout scheme can produce asymptotically smaller layouts than the

automata-based schemes for all languages, though it does not show that there aren't equally good schemes. There are languages, however, for which automata-based recognizers produce smaller layouts than expression based recognizers. A natural set of languages for which automata-based recognizers are superior to expression-based recognizers is the family $\{C_n\}$ of counting languages, defined by:

$$C_n = (0^n)^*.$$

C_n thus consists of strings of zeroes whose lengths are multiples of n. The shortest regular expression for C_n is the one given above, so that any expression-based recognizer requires at least $\Omega(n)$ area. (E^n is just a shorthand for $EE \ldots E$, where E is repeated n times.) The minimum-state deterministic automaton for C_n has only n states, however, so an automata-based recognizer of $O(\log n)$ area can be built using a counter. If area is to be minimized, automata-based recognizers such languages should be preferred to expression-based recognizers.

Recognizers based upon non-deterministic automata can have area advantages. Some layout algorithms, particularly those based upon PLA's, can benefit from the additional structure encoded in non-deterministic automata. The logic required to implement the state-transition function may be more regular than the corresponding logic in the minimal deterministic automaton, and so may be easier to lay out.

Any language that has a non-deterministic automaton with s states can be recognized using $O(s^2)$ area, using a PLA-based machine [28]. A network of PLA's can often significantly reduce the area required [75]. If a small number of compatible sets of states can be found, the use of multiple copies of deterministic machines can also produce small recognizers. An s-state machine with k compatible sets can be realized by k deterministic machines using only $O(ks \log s)$ total area.

Automata-based recognizers are not always smaller than expression-based recognizers. There are regular languages for which expression-based recognizers have minimal area.

Theorem 4-2: For any algorithm for layout of recognizers, and for any choice of n, there is some language with an n-character regular expression whose recognizer layout takes $\Omega(n)$ area.

Proof: Let E_n be the set of regular expressions consisting of n characters from {0, 1} concatenated together. E_2 is thus {00, 01, 11, 10}. Each of the 2^n expressions in E_n specifies a different language (with every language consisting of one string). As in the proof of Theorem 4-1, an algorithm for laying out every recognizer in E_n in less than area $\Omega(n)$ could be converted to a way of recording n bits in less than area $\Omega(n)$. Thus, no such algorithm exists.

□

The area bound of $\Omega(n)$ can be attained by either the systolic recognizer or the expression-tree recognizer described in Section 4.2. Theorem 4-2 shows that the areas of these recognizers are asymptotically optimal for some languages.

A natural family of languages for which expression-based recognizers have optimal area is the family $\{L_n\}$ defined by:

$$L_n = (0+1)^*1(0+1)^n.$$

That is, L_n includes all strings over $\{0, 1\}$ such that the $n+1$'st character from the end is a 1. A regular expression describing L_n has $3n+5$ operators and operands, so that an expression-based recognizer uses $3n+5$ cells. In fact, if the cells for set comparison that are discussed in Section 2.2 are used, only n cells are needed. Since concatenation is the dominant operation in the expressions L_n for large n, the recognizers can be laid out in linear area, even using the simple collinear layouts of Section 3.1. A deterministic automaton for L_n, on the other hand, must remember n bits, and thus has at least 2^n states. Even with the most compact encoding, $\Omega(n)$ area is required for the state register. The expression-based recognizers are thus among the smallest that can be made and should be used for languages of this type.

A monoid-based recognizer for a language that has an s-state non-deterministic automaton can be built using $O(s^2)$ area. The syntactic monoid is represented using $s \times s$ matrices, with multiplication performed in an $s \times s$ array of logic. However, it seems unlikely that monoid-based recognizers can be smaller than automata-based recognizers. Any small circuit for computing monoid products can be translated directly to a small circuit for computing state transitions. Any encoding of monoid elements can thus be translated directly into a state encoding, and the transition logic will require no more space than the logic used for monoid composition. Therefore, if small area is required, a silicon compiler should choose between automata-based and expression-based recognizers.

For some applications of language recognition, such as filtering in logic-per-track database machines and hardware monitoring, the speed of recognition is a major concern. While the speed of a recognizer is more dependent on details of implementation than is its area, some general guidelines can still be given. The recognizer circuits described in this thesis operate in discrete time steps, or beats. They read the input string one character at a time, performing some computation during the reading process. The speed of a recognizer is therefore determined by the speed at which it can read characters.

The fastest of the methods surveyed is monoid composition. Using this method, characters from

the input stream can be processed at high speed even if the circuit technology used is quite slow. An input stream of length n can be split into log n substrings of length $n/\log n$, and the substrings can be piped into a tree of composition elements with depth log n. In this way, the string of length n can be processed in the time taken for log n compositions. Even though n beats are still required to read the input, each beat can be log n/n as long as if each character were completely processed before reading the next character. Using monoid composition then, slow hardware can be used to process long, fast input streams.

The other methods surveyed require each character to be processed as it is read. In the automata-based methods, for example, enough time must pass between characters for the next state to be computed. This could be a memory cycle in a microprocessor implementation, or as many as log log s gate delays for a register and logic implementation of an s-state machine. The expression-based recognizers require enough time in each beat for signals to propagate in all data-paths. Since these methods are limited by delays in signal propagation and combinational logic, the systolic recognizer of Chapter 2 is a promising scheme for attaining high speed. This method avoids broadcasting the input string to all recognizers, so that the fanout and consequent data transfer time is small.

For most problems that are solved in VLSI, area and time can be traded. This leads to investigations of lower bounds on area-time products. For example, Brent and Goldschlager [10] have proven a lower bound of $\Omega(n^{1+\alpha})$ on $AT^{2\alpha}$, where $\alpha \in [0, 1]$, for determining whether a string of length n is in a context-free language. Such questions are not so interesting in regular language recognition, especially if the time to read the input is counted in the time taken by the circuit. The area of a regular language recognizer depends only on the language to be recognized (as opposed to a context-free recognizer, whose stack size depends on the input string). Only the time for recognition depends on the input string in a regular language recognizer. Any area-time lower bound on regular language recognition thus degenerates to a lower bound on time, which for the recognizers in this thesis is $\Omega(n)$.

A final basis for comparison of recognition algorithms is their suitability for extension. How easily can they be modified to perform tasks other than recognition? Such tasks are required in applications of regular language recognition. In applications such as backend database machines [5] and hardware monitors [9], output values are essential. In other applications such as text processing and lexical analysis, the input stream must be split into tokens or lexemes. The requirement for one of these auxiliary tasks may influence the selection of recognition algorithms for inclusion in a specialized silicon compiler.

As noted in Section 2.2, adding a disable signal to expression-based circuits allows them to partition the input stream into lexemes by stopping any recognition that is in progress. Similar modifications can be made to any automaton, by adding a reset signal that forces it to the start state. For monoid-based methods, however, the modifications are more difficult. The match result for a text string emerges from an n-cell fan-in tree $\log^* n$ beats after input of the final character, so that the input stream may need to back up to start recognizing the next token. This additional complexity seems to rule out the use of monoid-based fan-in trees in lexical analyzers. Either automata or expression based methods should be chosen.

Any of the recognizers can be modified to produce output values. The theory of finite-state automata with output is well developed [44] so that if the utmost flexibility in outputs is needed, programmable automata-based methods should probably be used. Of course, any pattern of outputs can be realized by a collection of expression-based recognizers, since the conditions for any state transition in an automaton can be written as a regular expression. Moreover, internal RES signals from recognizers are often meaningful, since they indicate the recognition of subexpressions. Although expression-based recognizers may be larger than equivalent automata-based recognizers, in some cases expression-based recognizers with auxiliary output seem to be natural and efficient for implementing controllers [76]. Individual applications and output requirements must be examined before a choice can be made.

This chapter has surveyed methods for recognizing regular languages. A silicon compiler that was specialized for language recognition might include more than one of these methods in its repertoire. By referring to the details of an application, a specialized compiler could choose a method meeting the requirements for area, speed, and additional functions. Inclusion of knowledge about the application area within the compiler leads to the selection of good methods and to the design of efficient chips.

Chapter 5
Syntax Directed Verification of Specialized Silicon Compilers

Many current VLSI designs are composed of standard cells, each of which performs a simple function but which are wired together to perform more complex tasks. Often it is not obvious that the function performed by cell combination is the one specified, even if the cells themselves are correct. Some means is therefore needed for proving the properties of large circuits made from small cells.

This chapter describes a syntax-directed technique for verifying the correctness of circuits composed from standard cells. This technique allows proofs of correctness to be developed in a mechanical way. It relies on the use of an attributed context-free grammar to specify both the function and structure of the legal combinations of cells. The terminal characters in the grammar correspond to the primitive cells, and non-terminals correspond to combinations of cells. The grammar's start symbol corresponds to the class of circuits whose correctness is to be verified. By proving a single theorem for each production in the grammar, the correctness of any circuit constructed according to the grammar may be verified.

Syntax-directed verification is particularly applicable to specialized silicon compilers. The lengths of proofs using this technique are independent of the size of the circuits, but depend only on the complexity of the grammar used by the compiler. Since specialized silicon compilers can be expected to use relatively simple grammars, correctness proofs will be short and comprehensible. A description of the verification method will be given, followed by several example correctness proofs of specialized silicon compilers.

5.1. The Verification Method

To prove the correctness of circuits built by a specialized silicon compiler, we must prove a theorem of this form:

- If all primitive cells are correct, then any legal combination of cells will be correct.

To construct this kind of theorem, we must give the specifications of the cells, tell what combinations of cells are legal, and give a rule for determining the form of any legal cell combination from its specifications. In this chapter the specifications of the cells are derived from the cell designs, and are treated as axioms. The legal combinations of cells are precisely those circuits that are generated by the attributed context-free grammar. Both the specification and structure of a circuit depend upon its derivation in the grammar. The cell designs and context-free grammar used in a specialized silicon compiler thus provide the basic components of the theorem to be proven.

As with program correctness, proof of circuit correctness proceeds in two steps: development of verification conditions, followed by their proof. To develop the verification conditions we make use of *syntactic assertions* on the values and timings of signals at the ports of each primitive cell and compound circuit. These assertions correspond to the inductive assertions [27] of program verification and may be thought of as specifications for the circuits. Each verification condition is a theorem relating the syntactic assertions of a compound circuit to those of its components.

One syntactic assertion is required for each symbol of the grammar. Terminal symbols of the grammar correspond to primitive cells, and the assertions for these symbols are simply the primitive cell specifications. Assertions for the non-terminals are specifications of the various compositions of primitive cells. The assertion for the start symbol is thus the specification for a complete circuit constructed using the grammar.

Once we have the syntactic assertions we can develop the verification conditions. Each production of the grammar corresponds to one verification condition, which states that the syntactic assertions of the symbols on the right side of the production imply the assertion on the left. In other words, the verification condition for a production ensures the correctness of the circuit on the left side of the production, as long as the circuits on the right side meet their specifications. Proof of these theorems, one for each production, completes the verification of the circuit family.

Notice that this verification technique requires that each production have only one symbol on the left, as in a context-free grammar. If all productions are of the form:

R → abcd
S → xRy

with only one non-terminal on the left, then an assertion for the non-terminal on the left in each production can be proved from the assertions for the symbols on the right. In the example above, an assertion for R can be proved from assertions for a, b, c and d, and an assertion for S can be proved from assertions for x, R and y.

Non-context-free grammars, on the other hand, have productions of the form:

pRq → abcd
S → xRy

in which more than one symbol appears on the left. It may be impossible to prove assertions for some of the non-terminals in such a grammar. In the example, assertions for pRq can be proved from assertions for a, b, c and d. Nothing at all can be proved about R itself, however, and hence nothing can be proved about S.

5.2. A Digital Filter Example

As an example of syntax-directed verification, we show that a specialized silicon compiler for constructing digital filters is correct. We begin by describing digital filters and a compiler for constructing them from small cells. We next derive the syntactic assertions for cells and for more complex circuits from the designs for the cells and the definition of digital filters. Finally, we construct and prove one of the verification conditions for the compiler.

A digital filter computes the solution to a linear recurrence of this form:

$$y_i = \sum_{0 \le j \le n-1} w_j x_{i-j} + \sum_{1 \le j \le n} r_j y_{i-j}.$$

That is, given an input sequence $\{x_i\}$ it computes an output sequence $\{y_i\}$ in which each term is a linear combination of preceding terms from the input and output sequences.

Kung [49] has shown how to build a linear pipeline for any digital filter of this type, using the cell shown in Figure 5-1. The cell stores two coefficients, *r* and *w*, and performs two multiplications and two additions. Figure 5-2 shows a linear pipeline constructed using this cell for the filter:

$$y_i = w_0 x_i + w_1 x_{i-1} + w_2 x_{i-2} + r_1 y_{i-1} + r_2 y_{i-2} + r_3 y_{i-3}.$$

Snapshots of the pipeline at two consecutive beats are shown in the figure. The sequence $\{x_i\}$ is input from the host at the left, one term every two beats, so that alternate cells in the pipe are idle. Data in this sequence moves rightward through the pipe, one cell per beat. The sequence $\{y_i\}$ is computed

by the pipeline; partial results move leftward toward the host, with a term of the form $wx + ry$ added in every cell. The small cell between the host and the pipeline is a unit delay that feeds completed y values back into the z input to the pipeline.

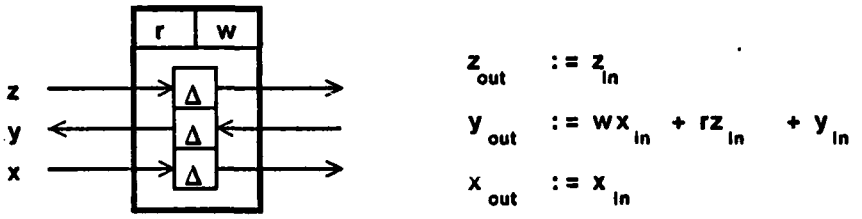


Figure 5-1: Cell for constructing digital filters

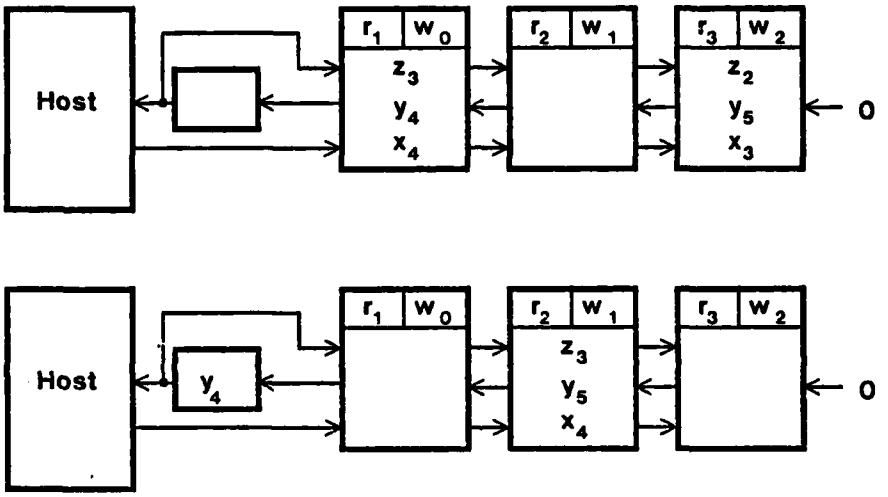


Figure 5-2: A digital filter pipeline

A specialized silicon compiler for digital filters takes as input the w and r coefficients and produces a line of cells with the coefficients in the right places. To prove that circuits constructed by this compiler actually realize the correct digital filters, we must first list the primitive cells and compound circuits, together with their syntactic assertions. Let the symbol c(r,w) denote a cell of the type shown in Figure 5-1, with stored weights r and w. If we let x_t, y_t and z_t be the data at the left ports of the cell at time t, and x'_t, y'_t and z'_t be the data at the right ports, the assertion $P_c(r,w)$ for c(r,w) is the conjunction of:

$$(\forall t)\, y_t = wx_{t-1} + rz_{t-1} + y'_{t-1} \qquad (5\text{-}1)$$
$$(\forall t)\, z'_t = z_{t-1}$$
$$(\forall t)\, x'_t = x_{t-1}.$$

The holding register at the left end of the pipe will be denoted by the symbol h. The cell shown in Figure 5-3, with two ports on the left and three on the right, can perform this function. The assertion P_h is the conjunction of:

$$(\forall t)\, z'_t = y_t = y'_{t-1} \qquad (5\text{-}2)$$
$$(\forall t)\, x'_t = x_t.$$

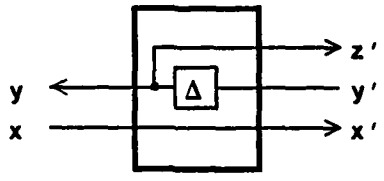


Figure 5-3: The cell h: a holding register

To provide the zero input on the right end of the pipe, we will construct a dummy end cell denoted by the symbol e, and shown in Figure 5-4. This cell just outputs 0 on its y output, so assertion P_e is:

$$(\forall t)\, y_t = 0. \qquad (5\text{-}3)$$

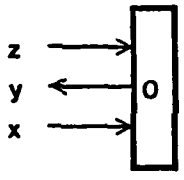


Figure 5-4: The cell e: a source of zeroes

Two kinds of compound circuits are used in constructing filters. The first is the filter itself and the second is a pipeline without the holding register. Let the symbol $F(r_1 \ldots r_n \;\; w_0 \ldots w_{n-1})$ denote a filter with weights $r_1 \ldots r_n\, w_0 \ldots w_{n-1}$. This type of circuit has an input port x and an output port y, both at the left. Since alternate cells in the pipe are idle, if x_i is input at time t, then at time $t+2$, x_{i+1} is input, and y_i is output. Thus the assertion for a filter states that the output at time t is a linear combination of the inputs and outputs at times $t-2$, $t-4$, ..., $t-2n$. Symbolically, $P_F(r_1 \ldots r_n\, w_0 \ldots w_{n-1})$ is:

$$(\forall t)\, y_t = \sum_{0 \le i \le n-1} w_i x_{t-2-2i} + \sum_{1 \le i \le n} r_i y_{t-2i}. \qquad (5\text{-}4)$$

The other kind of compound circuit is a pipeline without the holding register h, but with the end cell e. The symbol $P(r_1 \ \ldots \ r_n \ w_0 \ \ldots \ w_{n-1})$ denotes a pipeline with weights $r_1 \ \ldots \ r_n \ w_0 \ \ldots \ w_{n-1}$. This pipeline has two inputs, x and z, and one output, y, all at the left. Assertion $P_P(r_1 \ \ldots \ r_n \ w_0 \ \ldots \ w_{n-1})$ is:

$$(\forall t)\ y_t = \sum_{0 \le i \le n-1} w_i x_{t-1-2i} + \sum_{1 \le i \le n} r_i z_{t+1-2i}. \qquad (5\text{-}5)$$

Filters can be built from these cells using the attributed grammar:

1. $F(r_1 \ldots r_n \, w_0 \ldots w_{n-1}) \rightarrow hP(r_1 \ldots r_n \, w_0 \ldots w_{n-1})$
2. $P() \rightarrow e$
3. $P(r_1 \ldots r_n \, w_0 \ldots w_{n-1}) \rightarrow c(r_1 \, w_0)P(r_2 \ldots r_n \, w_1 \ldots w_{n-1})$.

The semantic actions corresponding to these productions are:

1. Hook the hold cell h to the left end of the pipe P to produce the filter F.
2. Use the end cell e as the pipe.
3. Hook the right port of the cell c to the left port of the pipe to produce a new pipe.

This compiler has three verification conditions, one for each production of the grammar. We state and prove the condition for production 3, which is the production that constructs long pipelines from short ones. The verification condition for this production states that the syntactic assertions for the cell (Assertion (5-1)) and pipeline (Assertion (5-5)) of Figure 5-5 imply the assertion for the new pipeline constructed by production 3. This means that if both the cell and pipeline of Figure 5-5 are correct, then the new pipeline obtained by hooking them together is also correct.

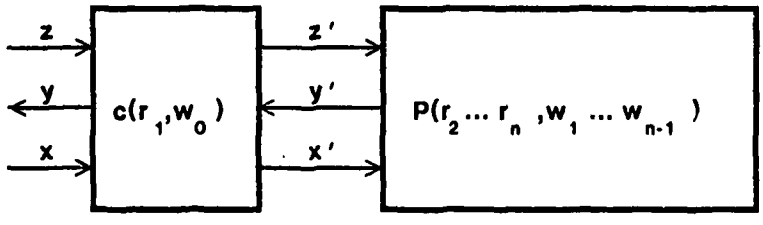


Figure 5-5: The effect of production 3

As shown in Figure 5-5, the right port of $c(r_1,w_0)$ is hooked to the left port of

$P(r_2 \ldots r_n\, w_1 \ldots w_{n-1})$. Thus, we can use the same symbols for the data on both, namely, $x'_t\, y'_t$ and z'_t. The assertion $P_P(r_2 \ldots r_n\, w_1 \ldots w_{n-1})$ is then:

$$(\forall t)\, y'_t = \sum_{0 \le i \le n-2} w_{i+1} x'_{t-1-2i} + \sum_{1 \le i \le n-1} r_{i+1} z'_{t+1-2i}.$$

Reindexing the sums, we obtain:

$$(\forall t)\, y'_{t-1} = \sum_{1 \le i \le n-1} w_i x'_{t-2i} + \sum_{2 \le i \le n} r_i z'_{t+2-2i}. \tag{5-6}$$

The verification condition for production 3 states that Assertion (5-6), together with the assertion $P_c(r,w)$ (Assertion (5-1)) imply the assertion $P_P(r_1 \ldots r_n\ w_0 \ldots w_{n-1})$ (Assertion (5-5)). To prove this, substitute z_{t+1-2i} for z'_{t+2-2i}, and x_{t-1-2i} for x'_{t-2i}, as permitted by the second and third conjuncts of Assertion (5-1). This substitution leads to this expression for y'_{t-1}:

$$(\forall t)\, y'_{t-1} = \sum_{1 \le i \le n-1} w_i x_{t-1-2i} + \sum_{2 \le i \le n} r_i z_{t+1-2i}.$$

Substitution of this expression into the first conjunct of Assertion (5-1) and rearrangement of sums results in Assertion (5-5), which shows that the verification condition is indeed satisfied. The verification conditions for productions 1 and 2 are proved in a similar manner.

Besides illustrating the verification technique, this example illustrates the use of an attributed context-free grammar to specify circuits. The productions in the filter grammar specify relations on the coefficients as well as grammatical transformations. To be used in a derivation, the production must satisfy constraints of both the grammar and the attributes. For example, $F(1, 2) \rightarrow hP(2, 3)$ is not a legal production in this grammar, since the attributes don't match. These attributed grammars can be thought of as schema that generate infinite context-free grammars. The essential property of context-free grammars is retained: each production has only one symbol on its left-hand side. The use of attributed grammars extends the number of circuits that can be specified without affecting the verification technique.

5.3. Verification of the Systolic Expression Compiler

As a second example of this technique, we will verify the compiler described in Chapter 2 by showing that circuits constructed using the primitive cells in that chapter recognize the correct regular expressions. This the proof in this section uses a modification of the grammar that was presented in Chapter 2. In the modified grammar, attributes have been added to the symbols of the grammar, and the productions have been rearranged slightly to make the verification clearer.

The grammar used for verification of the compiler is:

$P[\varphi] \rightarrow \varphi$ Use a new φ cell as the circuit for P.

$R[E] \rightarrow P[E]$ Terminate the left port of the circuit for P by connecting ENB_{out} to RES_{in}

R[<letter>] → <letter>
Use a new terminated comparator for R.

R[E<letter>] → R[E]<letter>
Connect the left port of a new comparator to the right port of R.

$R[E_1E_2] \rightarrow R[E_1]P[E_2]$
Connect the left port of P to the right port of R.

$P[(E_1 + E_2)] \rightarrow (R[E_1] + R[E_2])$
Connect the right ports of the R's to the top and bottom ports of a new or-node.

$P[(E)^*] \rightarrow (R[E])^*$.
Connect the right port of R to the top port of a new star-node.

Attributes in the grammar above are written in square brackets following the symbols to which they are attached. Symbols within the attributes represent symbols or subexpressions of the regular expression, while symbols outside the attributes represent cells or subcircuits. Thus, the + on the left-hand side of production 6 is a symbol in the regular expression $(E_1 + E_2)$, which is P's attribute, while the + on the right-hand side represents a primitive cell in the recognizer. Other than the addition of attributes, the only change in the grammar is the replacement of the production P→<letter> with the two productions R→<letter> and R→R<letter>. This does not change the circuits generated by the grammar; the only effect is to replace a single verification condition in the proof of correctness with two separate verification conditions. This replacement avoids a proof by cases by splitting the cases across the two verification conditions.

Two auxiliary truth functions, called INIT and **until**, are needed to state the syntactic assertions in this grammar. One of these, INIT(t, C), is true if and only if the circuit C has been initialized at time t. A comparator circuit is initialized when its ENB and RES registers contain 0, and a larger circuit is initialized when all of its components are initialized. The INIT function can therefore be defined inductively, using the grammar above. In this definition, and throughout this section, a signal name with a prime, such as ENB′, refers to that signal on the left port of a circuit.

Definition 5-1: For any circuit C, INIT(t, C) is defined by:

INIT(t, <letter>) ≜ $\neg RES'_t \wedge \neg ENB_t$

INIT(t, P[φ]) ≜ **true**

INIT(t, R[E]) ≜ INIT(t, P[E])

INIT(t, R[<letter>]) ≜ INIT(t, <letter>)

INIT(t, R[E]<letter>) ≜ INIT(t, R[E]) ∧ INIT(t, <letter>)

INIT(t, R[E_1]P[E_2]) ≜ INIT(t, R[E_1]) ∧ INIT(t, P[E_2])

INIT(t, R[E_1] + R[E_2]) ≜ INIT(t, R[E_1]) ∧ INIT(t, R[E_2])

INIT(t, P[(E)*]) ≜ INIT(t, R[E])

The second auxiliary function, **until**, describes the interaction of signals that change on the same beat. It is written in infix notation as (a_t **until** b_t), and its informal meaning is that within beat t, a_t is true at least as long as b_t is false. The beats are specific instants in time, and signals in the circuit change between beats. Correct values for beat t are attained in the interval between $t-1$ and t. The **until** function describes the order of signal transitions within a beat.

The reason for introducing the **until** function is to help describe the action of the clocked OR gates. Recall from Chapter 2 that clocked OR gates prevent latch-up in cycles of OR gates by outputting false for a brief time before each beat. The **until** function will be used to state that the outputs of cells containing clocked OR gates remain false as long as their inputs are false. This will ensure that no latch-up involving those cells occurs.

Before **until** can be defined formally, a description of the exact order of events within a beat is needed. For concreteness, the definition of **until** is based on the clocked OR gate shown in Figure 2-9 and the timing used in the E.T. chip (see Section 3.3). Each beat consists of four steps:

1. Begin setting up the inputs to the cell and lower φ_1;

2. Raise φ_2 and finish setting up the inputs to the cell (making at most one transition on each input);

3. Lower φ_2;

4. Raise φ_1 and hold it high until the next beat.

In this clocking scheme, every signal on beat t is either at its final value at the start of φ_2, or attains its final value through a single transition during φ_2. In fact, the only signals in any recognizer that make a transition during φ_2 are either the outputs of clocked OR gates, or result from passing those outputs through one or more OR gates. These signals are false at the start of φ_2 and may have a single transition to true during φ_2. Thus, any signal that appears in a recognizer is either at its final value at the start of φ_2 or makes a single transition from false to true.

Now that the timing of events within a beat has been described, a formal definition of **until** will be given. For signals a_t and b_t, $(a_t$ **until** $b_t)$ is defined to mean that, within φ_2 of beat t, a_t is true for at least as long as b_t has not yet attained a final value of true. According to this definition, $(a_t$ **until** $b_t)$ is true if a_t is true throughout φ_2, for instance. Notice that **until** operates on signals, not truth values. The truth value of a signal is its value exactly on the beat, but **until** operates on the signals themselves. It is left undefined except for pairs of signals on the same beat.

Throughout the proof of correctness, **until** will be used only in statements of the form $(\neg a_t$ **until** $b_t)$, where a_t and b_t are recognizer signals (such as RES_t or ENB_t). As observed above, a_t and b_t must either be at their final values at the start of φ_2 or make precisely one transition during φ_2 from false to true. Under this restriction, $(\neg a_t$ **until** $b_t)$ is true if and only if one of these conditions holds:

1. a_t is false (thus making no transition during φ_2),

2. b_t is true at the start of φ_2 (and remains true);

3. b_t becomes true during φ_2, and before a_t.

Other than substitution of equivalent signals, only one rule of inference involving **until** is needed in the correctness proof. This is,

$$\neg a_t \vdash (\neg a_t \text{ until } b_t),$$

which is the first of the conditions above. The other two conditions can be used to ensure that cells satisfy their syntactic assertions.

With both auxiliary functions defined, the syntactic assertions for systolic recognizer components can be constructed. Each symbol in the grammar has an associated syntactic assertion. Assertions for the non-terminal symbols, corresponding to compound circuits, will be presented first.

The syntactic assertion for a recognizer has two parts. The first part says that RES_t is true on a beat directly after any successful match, where the successful match is preceded by ENB. This is asserted regardless of the length of the matched string, so that if the circuit recognizes ϵ, RES_t is true at least as often as ENB_t is. This first part of the assertion is all that is required to ensure that the circuit acts as a recognizer on its own.

The second part of the assertion says that if RES_t turns on before ENB_t then the circuit must have matched a string of non-zero length. This ensures that the recognizer does not take part in a cycle of OR gates. Symbolically, ASSERT(R[E]) is:

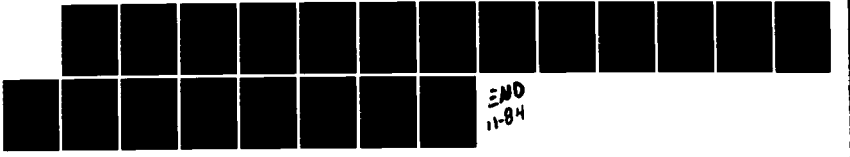
$$
\begin{aligned}
&(\forall t)\,(\forall b{>}0)\ \text{INIT}(t-2b,\ \text{R[E]}) \Rightarrow \qquad (5\text{-}7)\\
&\quad \{[\text{RES}_t \equiv (\exists n{\in}\{0\ldots b-1\})\,(\text{ENB}_{t-2n} \wedge \langle \text{CHR}_{t-2n+1}\cdots\text{CHR}_{t-1}\rangle \in \text{E})]\\
&\quad \wedge [((\neg \text{ENB}_t \ \textbf{until}\ \text{RES}_t) \wedge \text{RES}_t) \Rightarrow\\
&\qquad (\exists m{\in}\{1\ldots b-1\})(\text{ENB}_{t-2m} \wedge \langle \text{CHR}_{t-2m+1}\ldots\text{CHR}_{t-1}\rangle \in \text{E})]\}.
\end{aligned}
$$

One symbolic construction in this assertion deserves explanation. The symbol $\langle \text{CHR}_{t-2n+1} \cdots \text{CHR}_{t-1}\rangle$ stands for the string of characters that comes every two beats, starting with beat $t-2n+1$ and extending no later than beat $t-1$. If $n=0$, of course, this string is ϵ (the empty string).

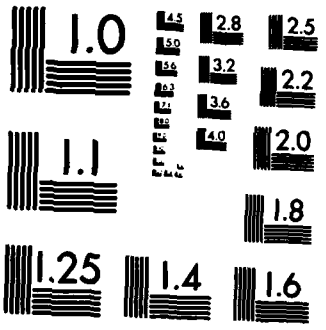
The other compound circuit that can be constructed using this grammar is the primitive recognizer, represented by the non-terminal P. Because of the modifications to the grammar, no primitive recognizer contains any delays between the left and right ports (although delays may be included in the subcircuits attached to the top and bottom ports). A primitive recognizer has signals ENB, RES and CHR at the right port, and ENB′, RES′ and CHR′ at the left port. In symbols, the assertion ASSERT(P[E]) for a primitive recognizer for E is the conjunction of:

AD-A146 526 SPECIALIZED SILICON COMPILERS FOR LANGUAGE RECOGNITION (U) CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE M J FOSTER JUL 84 CMU-CS-84-143 2/2

UNCLASSIFIED F33615-81-K-1539 F/G 9/2 NL



END
11-84



$$(\forall t)\ \mathrm{ENB}'_t = \mathrm{ENB}_t \tag{5-8}$$

$$(\forall t)\ \mathrm{CHR}'_t = \mathrm{CHR}_t$$

$$(\forall t)\ (\forall b{>}0)\ \mathrm{INIT}(t-2b, \mathrm{P}[\mathrm{E}]) \Rightarrow$$
$$\{[\mathrm{RES}_t = (\exists n{\in}\{0 \ldots b-1\})(\mathrm{RES}'_{t-2n} \wedge \langle \mathrm{CHR}_{t-2n+1} \ldots \mathrm{CHR}_{t-1}\rangle \in \mathrm{E})]$$
$$\wedge [((\neg \mathrm{RES}'_t \ \mathbf{until}\ \mathrm{RES}_t) \wedge \mathrm{RES}_t) \Rightarrow$$
$$(\exists m{\in}\{1 \ldots b-1\})(\mathrm{RES}'_{t-2m} \wedge \langle \mathrm{CHR}_{t-2m+1} \ldots \mathrm{CHR}_{t-1}\rangle \in \mathrm{E})]\}.$$

The first two conjuncts of this assertion state that ENB and CHR are passed through P unchanged, while the third conjunct states that P functions as a primitive recognizer.

The syntactic assertions of the cells state that they function according to the circuit diagrams in Figures 2-5, 2-6, 2-7, and 2-10. A character comparator for the character "X" obeys the assertion ASSERT(X):

$$(\forall t)\ \mathrm{ENB}'_{t+1} = \mathrm{ENB}_t \tag{5-9}$$

$$(\forall t)\ \mathrm{CHR}'_{t+1} = \mathrm{ENB}_t$$

$$(\forall t)\ (\mathrm{RES}_t = (\mathrm{RES}'_{t-1} \wedge (\mathrm{CHR}_{t-1} = \mathrm{X})).$$

The signals on the left port are indicated, as in Assertion (5-8), with a prime. The first two conjuncts of ASSERT(X) simply state that ENB and CHR are passed through with unit delay, while the third conjunct describes the character comparison circuit.

The assertion for the φ cell merely states that it transmits CHR and ENB unchanged and outputs 0 on RES:

$$(\forall t)\ \neg \mathrm{RES}_t \tag{5-10}$$

$$(\forall t)\ \mathrm{CHR}_t = \mathrm{CHR}'_t$$

$$(\forall t)\ \mathrm{ENB}_t = \mathrm{ENB}'_t.$$

For the OR node, the upper and lower ports are denoted by the superscripts u and l, so that CHR^l is the character output of the upper port. The assertion ASSERT(+) is then the conjunction of:

With both auxiliary functions defined, the syntactic assertions for systolic recognizer components can be constructed. Each symbol in the grammar has an associated syntactic assertion. Assertions for the non-terminal symbols, corresponding to compound circuits, will be presented first.

The syntactic assertion for a recognizer has two parts. The first part says that RES_t is true on a beat directly after any successful match, where the successful match is preceded by ENB. This is asserted regardless of the length of the matched string, so that if the circuit recognizes ϵ, RES_t is true at least as often as ENB_t is. This first part of the assertion is all that is required to ensure that the circuit acts as a recognizer on its own.

The second part of the assertion says that if RES_t turns on before ENB_t then the circuit must have matched a string of non-zero length. This ensures that the recognizer does not take part in a cycle of OR gates. Symbolically, ASSERT(R[E]) is:

$$(\forall t)\,(\forall b{>}0)\,\text{INIT}(t-2b,\,\text{R[E]}) \Rightarrow \tag{5-7}$$
$$\{[\text{RES}_t \equiv (\exists n{\in}\{0\ldots b-1\})\,(\text{ENB}_{t-2n} \wedge \langle \text{CHR}_{t-2n+1}\ldots\text{CHR}_{t-1}\rangle \in \text{E})]$$
$$\wedge\,\{((\neg\text{ENB}_t \text{ until } \text{RES}_t) \wedge \text{RES}_t) \Rightarrow$$
$$(\exists m{\in}\{1\ldots b-1\})(\text{ENB}_{t-2m} \wedge \langle \text{CHR}_{t-2m+1}\ldots\text{CHR}_{t-1}\rangle \in \text{E})]\}.$$

One symbolic construction in this assertion deserves explanation. The symbol $\langle \text{CHR}_{t-2n+1} \ldots \text{CHR}_{t-1}\rangle$ stands for the string of characters that comes every two beats, starting with beat $t-2n+1$ and extending no later than beat $t-1$. If $n=0$, of course, this string is ϵ (the empty string).

The other compound circuit that can be constructed using this grammar is the primitive recognizer, represented by the non-terminal P. Because of the modifications to the grammar, no primitive recognizer contains any delays between the left and right ports (although delays may be included in the subcircuits attached to the top and bottom ports). A primitive recognizer has signals ENB, RES and CHR at the right port, and ENB′, RES′ and CHR′ at the left port. In symbols, the assertion ASSERT(P[E]) for a primitive recognizer for E is the conjunction of:

$$(\forall_t)\ \mathrm{RES}_t = \mathrm{RES}^l_t \vee \mathrm{RES}^u_t \tag{5-11}$$

$$(\forall t)\ \mathrm{CHR}_t = \mathrm{CHR}'_t = \mathrm{CHR}^l_t = \mathrm{CHR}^u_t$$

$$(\forall t)\ \mathrm{ENB}^l_t = \mathrm{ENB}^u_t = \mathrm{RES}'_t$$

$$(\forall t)\ \mathrm{ENB}'_t = \mathrm{ENB}_t$$

$$(\forall t)\ (\neg \mathrm{RES}'_t\ \textit{until}\ \mathrm{RES}_t) \wedge \mathrm{RES}_t \Rightarrow$$
$$[(\neg \mathrm{ENB}^l_t\ \text{until}\ \mathrm{RES}^l_t) \wedge \mathrm{RES}^l_t] \vee [(\neg \mathrm{ENB}^u_t\ \text{until}\ \mathrm{RES}^u_t) \wedge \mathrm{RES}^u_t].$$

The final conjunct in ASSERT(+) captures the direction of the OR gate in Figure 2-7. The output of the gate doesn't turn on until at least one of the inputs does. The conjunct states this indirectly, by relating the gate inputs and output to the single net containing the signals RES'_t, ENB^l_t, and ENB^u_t. The conjunct states that if the gate output turns on before the net does, then at least one of the inputs must also turn on before the net. The output therefore does not turn on before one of the inputs does.

Using the same convention for the upper port, ASSERT(•) is:

$$(\forall t)\ \mathrm{RES}_t = \mathrm{ENB}^u_t = \mathrm{RES}'_t \vee \mathrm{RES}^u_t \tag{5-12}$$

$$(\forall t)\ \mathrm{CHR}_t = \mathrm{CHR}'_t = \mathrm{CHR}^u_t$$

$$(\forall t)\ \mathrm{ENB}_t = \mathrm{ENB}'_t$$

$$(\forall t)\ (\neg \mathrm{RES}'_t\ \text{until}\ \mathrm{RES}_t) \Rightarrow (\neg \mathrm{ENB}^u_t\ \text{until}\ \mathrm{RES}^u_t)$$

$$(\forall t)\ (\neg \mathrm{RES}'_t\ \text{until}\ \mathrm{RES}_t) \wedge \mathrm{RES}_t \Rightarrow \mathrm{RES}^u_t.$$

Since the modified grammar for this compiler has six productions, there are six verification conditions. We will prove one that corresponds to the production P[(E)*]→(R[E])*, whose action is illustrated in Figure 5-6. This is the most interesting of the productions, since it clearly illustrates the use of the until function.

The verification condition for a production says that the syntactic assertion for the symbol on the

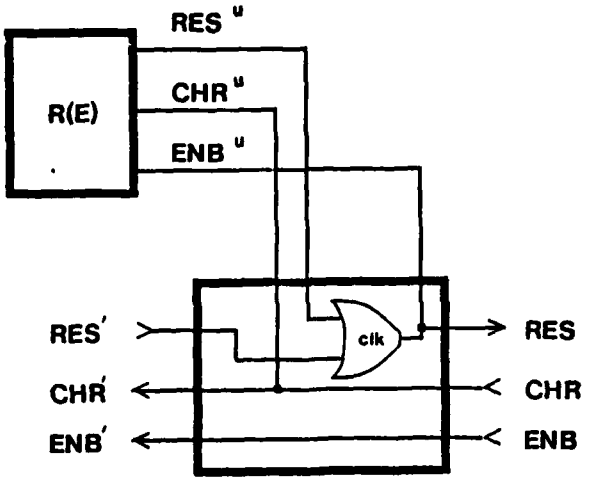


Figure 5-6: A primitive recognizer for (E)*

left hand side follows from the syntactic assertions for the symbols on the right hand side. To prove the verification condition for the production P[(E)*]→(R[E])*, we assume ASSERT(R[E]) and ASSERT(•) (Assertions (5-7) and (5-12)), apply the semantic action for the production, and prove ASSERT(P[(E)*]) (Assertion (5-8) with (E)* substituted for E). The semantic action is to connect the right port of R[E] to the upper port of the star node, as shown in Figure 5-6. Renaming signals in Assertion (5-7) to reflect the connection of R[E] to the upper part of the star node, we obtain:

$$
\begin{aligned}
&(\forall t)\,(\forall b{>}0)\ \mathrm{INIT}(t-2b,\ \mathrm{R[E]}) \Rightarrow \qquad (5\text{-}13)\\
&\quad \{[\mathrm{RES}^u{}_t \equiv (\exists n{\in}\{0 \ldots b-1\})(\mathrm{ENB}^u{}_{t-2n} \wedge \langle \mathrm{CHR}^u{}_{t-2n+1} \ldots \mathrm{CHR}^u{}_{t-1}\rangle \in \mathrm{E})]\\
&\quad \wedge [((\neg \mathrm{ENB}^u{}_t \ \mathbf{until}\ \mathrm{RES}^u{}_t) \wedge \mathrm{RES}^u{}_t) \Rightarrow\\
&\qquad (\exists m{\in}\{1 \ldots b-1\})(\mathrm{ENB}^u{}_{t-2m} \wedge \langle \mathrm{CHR}^u{}_{t-2m+1} \ldots \mathrm{CHR}^u{}_{t-1}\rangle \in \mathrm{E})]\}.
\end{aligned}
$$

The first two conjuncts in Assertion (5-8) follow from the second and third conjuncts of Assertion (5-12), so that all that remains is to prove the third conjunct. Assertion (5-13) and Assertion (5-12) will be used to prove this conjunct, using natural deduction. Choose fixed values for t and b and assume that INIT($t-2b$, P[(E)*]) is true. Then, by definition of INIT, INIT($t-2b$, R[E]), which is the antecedent of the third conjunct of Assertion (5-8), is also true. We must therefore prove the consequent of this conjunct:

$$[\text{RES}_t \equiv (\exists n \in \{0 \ldots b-1\})(\text{RES}'_{t-2n} \wedge \langle \text{CHR}_{t-2n+1} \ldots \text{CHR}_{t-1} \rangle \in E^*)]$$
$$\wedge [((\neg \text{RES}'_t \text{ until } \text{RES}_t) \wedge \text{RES}_t) \Rightarrow$$
$$(\exists m \in \{1 \ldots b-1\})(\text{RES}'_{t-2m} \wedge \langle \text{CHR}_{t-2m+1} \ldots \text{CHR}_{t-1} \rangle \in E^*)].$$

First, we prove the equivalence in the reverse direction. Suppose $(\exists n \in \{0 \ldots b-1\})$ $(\text{RES}'_{t-2n} \wedge \langle \text{CHR}_{t-2n+1} \ldots \text{CHR}_{t-1} \rangle \in E)]$. If $n=0$ then RES'_t is true, so RES_t follows from the first conjunct of Assertion (5-12). Otherwise, $n>0$, so by definition of E^* there is some finite sequence $\{t_i\}$:

$$t-2n = t_1 < t_2 < \ldots < t_k = t,$$

such that

$$(\forall i \in \{2 \ldots k\}) \langle \text{CHR}_{t_{i-1}+1} \ldots \text{CHR}_{t_i-1} \rangle \in E.$$

By supposition, RES'_{t_1} is true, hence so are RES_{t_1} and $\text{ENB}^u_{t_1}$, by Assertion (5-12). According to Assertion (5-13), because $\text{ENB}^u_{t_1}$ is true and $\langle \text{CHR}_{t_1+1} \ldots \text{CHR}_{t_2} \rangle \in E$, RES_{t_2} and $\text{ENB}^u_{t_2}$ are also true. By induction on k, so are $\text{ENB}^u_{t_k}$ and RES_{t_k} (which is RES_t). This proves the reverse equivalence.

We next prove the equivalence in the forward direction, by finding some $n \geq 0$ that makes the consequent true. Suppose RES_t is true. Then so is either RES'_t or $\neg\text{RES}'_t$. If RES'_t then the consequent is true with $n=0$. Otherwise, if $\neg\text{RES}'_t$ then RES^u_t is true, since $\text{RES}_t \equiv \text{RES}'_t \vee \text{RES}^u_t$. Furthermore, $\neg\text{RES}'_t$ implies $(\neg\text{RES}'_t \text{ until } \text{RES}_t)$. Thus, from clause 4 of Assertion (5-12), $(\neg\text{ENB}^u_t \text{ until } \text{RES}^u_t)$ is true. Merging these consequences of $\neg\text{RES}'_t$, we obtain $(\neg\text{ENB}^u_t \text{ until } \text{RES}^u_t)$ $\wedge \text{RES}^u_t$.

From Assertion (5-13), we can now conclude $(\exists m \in \{1 \ldots b-1\})(\text{ENB}^u_{t-2m} \wedge \langle \text{CHR}^u_{t-2m+1} \ldots \text{CHR}^u_{t-1} \rangle \in E)$. Let $t_1 = t-2m$. Since $m>0$, $t_1 < t$. Now note that $\text{ENB}^u_{t_1}$ is true, so RES_{t_1} follows from assertion (5-12).

The argument of the preceding two paragraphs can be repeated to show that either RES'_{t_1} or $(\exists t_2 < t_1)(\text{ENB}^u_{t_2} \wedge \langle \text{CHR}^u_{t_2+1} \ldots \text{CHR}^u_{t_1-1} \rangle \in E)$ is true. In fact, by induction, a finite sequence $\{t_k\}$ with

$$t-2b < t_k < t_{k-1} < \ldots < t_1 < t$$

can be found such that for each i, either RES'_{t_i} or $\text{ENB}^u_{t_{i+1}} \wedge \langle \text{CHR}^u_{t_{i+1}+1} \ldots \text{CHR}^u_{t_i-1} \rangle \in E$ is true. (The sequence is finite, because b is finite.) In this sequence, RES'_{t_k} must be true, since otherwise the sequence would have one more term. Set $n=(t-t_k)/2$, so that $t_k = t-2n$, and $n \in \{0 \ldots b-1\}$. We have just shown that RES'_{t-2n} is true. By the definition of E^*, $\langle \text{CHR}_{t-2n+1} \ldots \text{CHR}_{t-1} \rangle \in E^*$, since we have shown that $\langle \text{CHR}_{t-2n+1} \ldots \text{CHR}_{t_{k-1}-1} \rangle$, $\langle \text{CHR}_{t_{k-1}+1} \ldots \text{CHR}_{t-2} \rangle$, $\ldots$, $\langle \text{CHR}_{t_2+1} \ldots \text{CHR}_{t_1-1} \rangle$, $\langle \text{CHR}_{t_1+1} \ldots \text{CHR}_{t-1} \rangle$ are all in E. This completes the proof of the equivalence.

The second part of the verification condition is a proof of the implication. This is similar to the proof of the forward equivalence. Suppose $(\neg \text{RES}'_t \text{ until } \text{RES}_t) \wedge \text{RES}_t$. We must find an $m \in \{1 \ldots b-1\}$ such that $(\text{RES}'_{t-2m} \wedge \langle \text{CHR}_{t-2m+1} \ldots \text{CHR}_{t-1} \rangle \in E^*)$. From the assumption and from Assertion (5-12), we conclude $(\neg \text{ENB}^u_t \text{ until } \text{RES}^u_t)$ and RES^u_t. From Assertion (5-13) then, $(\exists p \in \{1 \ldots b-1\})\ (\text{ENB}^u_{t-2p} \wedge \langle \text{CHR}^u_{t-2p+1} \ldots \text{CHR}^u_{t-1} \rangle \in E)$. Since ENB^u_{t-2p} is true, Assertion (5-12) implies $\text{RES}'_{t-2p} \vee \text{RES}^u_{t-2p}$. If RES'_{t-2p}, just set $m=p$. Otherwise, $\neg \text{RES}'_{t-2p}$ lets us conclude $(\neg \text{RES}'_{t-2p} \text{ until } \text{RES}^u_{t-2p}) \wedge \text{RES}^u_{t-2p}$. As before, we can find a $q>p$ that lets the whole argument repeat. But, as in the forward equivalence, this can repeat only finitely many times, since b is finite. We set m to the largest of these indices, and the implication is proven. This concludes the proof of the verification condition for $P[(E^*)] \rightarrow (R[E]^*)$.

□

Proofs of the other verification conditions are similar in structure. The syntactic assertions for the symbols on the right side of each production are assumed, and the assertion for the left side is proven from them. The semantic actions provide a renaming of some of the signals in the assertions on the right side. Although there are many details to check, the proofs are essentially trivial.

5.4. Summary

This chapter has introduced a method for verifying properties of circuits composed from standard cells using a context-free grammar, as might be done by a specialized silicon compiler. The examples in this chapter show how proof of a small number of theorems can verify the correctness of any circuit built by the compiler. They also show that syntax-directed verification is applicable to non-trivial circuits, and is a worthwhile addition to the validation methods currently in use.

Despite its usefulness, syntax-directed verification should not be the sole validation method applied to specialized silicon compilers. One reason is that the correctness of the cells must be verified independently, since the syntax-directed technique depends on correct cells. Another is that syntax-directed verification shares many of the disadvantages of classical program verification using assertions [22]. Proofs tend to be long and detailed, but essentially trivial. It is as easy to err in constructing the proofs as in constructing the compilers. Moreover, syntactic assertions seem to be as difficult to construct as the inductive assertions used in verifying while loops. Without mechanical aids such as theorem provers, syntax-directed verification is probably not worthwhile. In any case, it should be augmented with other validation techniques, such as simulation of the circuits constructed by the compiler.

Although syntax-directed verification is not a panacea, it can aid in the design of correct specialized silicon compilers. If mechanical aids are available, correctness proofs can be developed at the same time as the design of the cells and grammar, to provide assurance that the compiled circuits will meet their specifications. Because of the usefulness of this technique, it is worthwhile to try to specify interconnections of standard cells using a context-free grammar. Designers of specialized silicon compilers should apply syntax-directed verification techniques to help ensure that circuits built with their systems will work as expected.

Chapter 6
Conclusions and Directions

This thesis has explored the construction of specialized silicon compilers, using a regular expression compiler as an example. The exploration has shown that specialized silicon compilers can be useful, feasible, and verifiable. The usefulness of the compiler comes from its ability to produce efficient chips automatically. Feasibility is shown in the thesis by the construction of both the compiler and a programmable layout that serves as a target for the compiler. Verification techniques similar to the one presented here can be used to prove the correctness of specialized silicon compilers. Because of the benefits of specialization shown in this thesis, specialized silicon compilers for other areas should be built using the same techniques.

Specialization in silicon compilers can greatly improve the chips that they produce. Because the compilers are specialized for a particular task domain, the compiled chips can make efficient use of area and time. For example, the regular expression compiler discussed here uses carefully designed primitive cells that are tailored to pattern recognition. The chips that it produces use a novel systolic algorithm that depends heavily on the description of patterns by regular expressions. finally, it uses layout schema that are particularly efficient for the circuits that it produces. None of these area and time saving techniques could be used without specialization to one area of application. Thus, specialization is a promising technique which lets silicon compilers make the proper tradeoffs between methods and compete with hand layout.

Specialization in programmable layouts is also beneficial. Both the cells and interconnections used in programmable layouts can be more efficient when their area of application is limited. Compact cells can be designed that are programmable for exactly the functions needed in the application area. Programmable interconnections can also be made more compact when the application area is fixed, since only limited forms of interconnection may be required. Programmable structures like the cutbus described in Chapter 3, that replace transistors with fixed connections, can be used only when the form of interconnections is known in advance. Application knowledge makes the benefits of programmable layouts available without exacting large costs in area or time.

The results of Chapter 5 show the advantages of formalized interconnection rules in specialized silicon compilers. The context-free grammar that was used to specify the interconnection of cells allows verification of the correctness of the compiler. It is important to note that the grammar need not be a finite context-free grammar, as shown by the filter example in Chapter 5. Structures such as rectangular arrays that cannot be described by finite context-free grammars may still have an infinite grammar of this type. The essential feature of these grammars is that each production has only one symbol on the left side. If an infinite set of productions can be handled by a finite set of schema, as in the filter example, then syntax directed verification can be applied. Formal composition rules of this type are beneficial and widely applicable.

In addition to their use in verification, interconnection rules structured as finite or infinite context-free grammars help make specialized silicon compilers extensible as well. Silicon compilers whose cells are composed according to a grammar can often be extended to use new cells by simply adding a few new productions to the grammar. By extension, silicon compilers can grow gradually, thus increasing their areas of application.

Further research is needed to extend this approach to other specialized silicon compilers. Compilers such as FIRST [7] should be expanded to accept a behavioral, rather than structural, specification of the signal processing chip. Ideally, the specification would be translated using a context-free grammar, as in this thesis. This would produce a modular, extensible, and verifiable compiler. Further specialization might produce compilers tailored for such applications as image processing, speech processing, or radar processing, all of which require slightly different types of signal processing operators. Outside of the signal processing area, several types of silicon compilers are possible. A very simple compiler could be built for clock-and-counter based circuits, for example. It might be possible to generate automatically such chips as UART's, interval timers, rate monitors, and frequency generators. These sorts of circuits are popular candidates for VLSI implementation [18], so that a specialized compiler for producing them could be useful. Another area in which specialized silicon compilers could be used is the construction of microprocessors. The MacPitts compiler [70, 72] is a promising step; it may be thought of as a specialized silicon compiler for microprocessors. Current research on generating high-quality data paths and controllers should be integrated with a translator for high-level descriptions similar to MacPitts. It might even be possible to build a compiler for analog circuits, so that a naive designer could combine operational amplifiers, sensors, and A/D and D/A converters to create analog subsystems. Each of these kinds of specialized silicon compilers would greatly ease the design of VLSI chips, so that research into their construction should prove profitable.

Some of these types of specialized compilers might require construction rules of a different form from that of the attributed context-free grammars studied here. Other methods of specifying a translation from behavioral descriptions to layouts should be identified and studied. The benefits of the method used here — modularity and comprehensibility — should be retained.

Programmable layouts for language recognition also deserve more research. Many of the problems dealing with cutbus layout are still open. For example, no efficient algorithms are known for cutbus layouts in which an edge may pass through some finite number of the cuts on a channel. Nor are similar structures known that route each edge through a constant number of switches, without requiring nodes to be collinear. If such structures were found, they could reduce the areas of soft programmable layouts significantly. If these problems were solved, a soft programmable layout could be a useful product in itself. Programmable cells and channels could be constructed for use in some specific application, such as hardware monitoring. Soft programmable chips of this type could eventually take over many functions for which custom hardware is currently used.

Even with the improvements offered by future research, specialized silicon compilers will not solve all problems of VLSI design. One problem is that specialized compilers are large, complex pieces of software. Specialized compilers are suitable only for application areas in which many different chips are needed, so that the design time for the compiler can be amortized over many chips. If only one or two chips need to be designed, use of more general tools is indicated, even though the design time may be longer. A second problem is that in many application areas, efficiency of chips is not an overriding concern. In these areas more general silicon compilers, or semicustom layouts, might be better choices for design aids. Only application areas in which many efficient custom chips must be designed quickly are suitable for specialized silicon compilers. Still a third problem is integration of systems. Specialized silicon compilers can produce efficient pieces of a system, but other tools are needed to interconnect these pieces. Specialized silicon compilers must be used together with other tools to build complete systems.

Despite their drawbacks, specialized silicon compilers can be useful tools. By exploiting the special characteristics of individual task domains, these compilers can produce efficient, automatically designed chips. The reduction in design complexity and increase in efficiency provided by specialized silicon compilers will help fulfill the potential of custom VLSI.

Chapter 7
Bibliography

[1] Aho, A. V.
Pattern Matching in Strings.
In Ronald V. Book (editor), *Formal Language Theory: Perspectives and Open Problems*, pages 325-348. Academic Press, New York, 1980.

[2] Anceau, F.
CAPRI: A Design Methodology and Silicon Compiler for VLSI Circuits.
In R. Bryant (editor), *Proceedings of the Third Caltech Conference on Very Large Scale Integration*, pages 15-32. Caltech, Pasadena, Ca., March, 1983.

[3] Andler, S.
Predicate Path Expressions: A High-Level Synchronization Mechanism.
PhD thesis, Carnegie-Mellon University, Computer Science Department, 1979.

[4] Backhouse, R. C.
Specification and Proof of a Regular Language Recognizer in Synchronous CCS.
Technical Report CSM-53, University of Essex, January, 1983.

[5] Bancilhon, F. and M. Scholl.
Design of a Backend Processor for a Data Base Machine.
In *Proceedings of International Conference on Management of Data*, pages 93-93g. ACM Sigmod, May, 1980.

[6] Beke, H. and W. Sansen.
CALMOS: A Portable Software System for the Automatic and Interactive Layout of MOS LSI.
In *Sixteenth Design Automation Conference.* I.E.E.E, June, 1979.

[7] Bergmann, N.
A Case Study of the F.I.R.S.T. Silicon Compiler.
In R. Bryant (editor), *Proceedings of the Third Caltech Conference on Very Large Scale Integration*, pages 413-430. Caltech, Pasadena, Ca., March, 1983.

[8] Bhatt, S. N. and C. E. Leiserson.
How to Assemble Tree Machines.
In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, pages 77-84. ACM, May, 1982.

[9] Bisiani, R., M. Annaratone, and M. J. Foster.
An Architecture for Real Time Debugging of Custom VLSI Chips.
In *1983 International Symposium on VLSI Technology, Systems and Applications.* March, 1983.

[10] Brent, R. P. and L. M. Goldschlager.
Area-Time Tradeoffs for VLSI Circuits.
In *Microelectronics '82*, pages 52-56. The Institution of Engineers, Australia, May, 1982.

[11] Brent, R.P. and H. T. Kung.
On the Area of Binary Tree Layouts.
Information Processing Letters 11(1):46-48, August, 1980.

[12] Bruegge, B. and P. Hibbard.
Generalized Path Expressions: A High Level Debugging Mechanism.
In *Proceedings of ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, pages 34-44. March, 1983.
Also appears in SIGPLAN Notices of August, 1983.

[13] Budzinski, R., J. Linn, and S. Thatte.
A Restructurable Integrated Circuit for Implementing Programmable Digital Systems.
In C. L. Seitz (editor), *Proceedings of the Second Caltech Conference on Very Large Scale Integration*, pages 481-508. Caltech, Pasadena, Ca., January, 1981.

[14] Campbell, R. H. and Habermann, A. N.
The Specification of Process Synchronization by Path Expressions.
In G. Goos and J. Hartmanis (editor), *Lecture Notes in Computer Science*, pages 89-102. Springer Verlag, New York, 1974.

[15] Chu, K. H. and K. S. Fu.
VLSI Architectures for High Speed Recognition of General Context-Free Languages and Finite-State Languages.
Technical Report TR-EE 81-42, Purdue University, November, 1981.

[16] Chung, F. R. K., F. T. Leighton, and A. L. Rosenberg.
Diogenes: A Methodology for Designing Fault Tolerant VLSI Processor Arrays.
Technical Report CS-1983-5, Department of Computer Science, Duke University, 1983.

[17] Cohen, D. and G. Lewicki.
MOSIS -- The ARPA Silicon Broker.
In *Proceedings of the Second Caltech Conference on VLSI.* California Institute of Technology, January, 1981.

[18] Conway, L., A. Bell, M. Newell, R. Lyon and Richard Pasco.
Implementation Documentatin for the MPC79 Multi-University Multiproject Chip-Set.
Filed on [Parc-Maxc]<Conway>MPC79imp.doc.
January 1, 1980.

[19] Crochiere, R. E. and A. V. Oppenheim.
Analysis of Linear Digital Networks.
Proceedings of the IEEE 63(4):581-595, April, 1975.

[20] Culik, K. and H. Jürgensen.
Programmable Finite Automata for VLSI.
International Journal of Computer Mathematics 14:259-275, 1983.
An earlier version appears as Technical Report CS-82-84 from the Computer Science Department, University of Waterloo.

[21] Culik, K., A. Salomaa, and D. Wood.
VLSI Systolic Trees as Acceptors.
Technical Report CS-81-32, Faculty of Mathematics, University of Waterloo, November, 1981.

[22] De Millo, R. A., R. J. Lipton, and A. J. Perlis.
Social Processes and Proofs of Theorems and Programs.
Communications of the ACM 22(5):271-280, May, 1979.

[23] Denyer, P. B. and D. J. Myers.
Carry-Save Arrays for VLSI Signal Processing.
In J. P. Gray (editor), *VLSI 81*, pages 151-160. University of Edinburgh, University of Edinburgh, August, 1981.

[24] Donze, R. L. and G. Sprozynski.
Masterimage Approach to VLSI Design.
Computer 16(12):18-25, December, 1983.

[25] Dowell, R., A.R. Newton and D.O. Pederson.
SPICE VAX User's Guide
Berkeley, California, 1979.

[26] Feldman, S. I.
The Circuit Design Language Xi (Ξ).
In *Proceedings of the IEEE International Conference on Computer Design*, pages 652-655. IEEE, October, 1983.

[27] Floyd, R.W.
Assigning Meanings to Programs.
In *Proceedings of the American Mathematical Society Symposium in Applied Mathematics*, pages 19-31. American Mathematical Society, 1967.

[28] Floyd, R. W. and J. D. Ullman.
The Compilation of Regular Expressions into Integrated Circuits.
JACM 29(3):603-622, July, 1982.

[29] Foster, M.J. and Kung, H.T.
The Design of Special-Purpose VLSI Chips.
Computer 13(1):26-40, January, 1980.
Reprint of the paper appears in *Digital MOS Integrated Circuits*, edited by Elmasry, M.I., IEEE Press Selected Reprint Series, 1981, pp. 204-217. A preliminary version of the paper, entitled "Design of Special-Purpose VLSI Chips: Example and Opinions," also appears in *Proceedings of the 7th International Symposium on Computer Architecture*, pp. 300-307, La Baule, France, May 1980.

[30] Foster, M. J. and H. T. Kung.
Recognize Regular Languages With Programmable Building Blocks.
Journal of Digital Systems 6(4):323-332, 1982.
A preliminary version of this paper appears in the VLSI-81 Proceedings, edited by John P. Gray.

[31] Fox, J. R.
The MacPitts Silicon Compiler: A View From the Telecommunications Industry.
VLSI Design 4(3):30-37, May/June, 1983.

[32] Garey, M. R., R. L. Graham, D. S. Johnson, and D. E. Knuth.
Complexity Results for Bandwidth Minimization.
SIAM Journal of Applied Mathematics 34:477-495, 1978.

[33] Hardage, K.
ASAP: Advanced Symbolic Artwork Preparation.
Lambda 1(3):32-39, Fourth Quarter, 1980.

[34] Haskin, Roger Lee.
Hardware for Searching Very Large Text Databases.
PhD thesis, University of Illinois at Urbana-Champaign, 1980.

[35] Hitchcock, C. Y. and D. E. Thomas.
A Method of Automatic Data Path Synthesis.
In *Proceedings of the 20'th Design Automation Conference.* IEEE, June, 1983.

[36] Holzmann, G. J.
The Analysis of Message Passing Systems.
Computing Science Technical Report 93, Bell Laboratories, January, 1981.

[37] Hopcroft, J.E. and J.D. Ullman.
Introduction to Automata Theory, Languages, and Computation.
Addison-Wesley Publishing Co., 1979.

[38] Huffman, D. A.
The Synthesis of Sequential Switching Circuits.
Journal of the Franklin Institute 257(3-4):161-190, 1954.

[39] Hunt, H. B.
The Equivalence Problem for Regular Expressions with Intersection is not Polynomial in Tape.
Technical Report TR 73-156, Department of Computer Science, Cornell University, 1973.

[40] Jarvis, J. F.
Feature Recognition in Line Drawings Using Regular Expressions.
In *Third International Joint Conference on Pattern Recognition.* IEEE, 1976.

[41] Johannsen, D.
Bristle Blocks, A Silicon Compiler.
In *Proceedings of the Caltech Conference on Very Large Scale Integration*, pages 303-310. January, 1979.

[42] Karlin, A. R., H. W. Trickey, and J. D. Ullman.
Experience with a Regular Expression Compiler.
In *Proceedings of the International Conference on Computer Design*, pages 656-665. IEEE, October, 1983.

[43] Kleene, S. C.
Representation of Events in Nerve Nets and Finite Automata.
In C. Shannon (editor), *Automata Studies*, chapter 1, pages 3-41. Princeton University Press, Princeton, N. J., 1956.

[44] Kohavi, Zwi.
Switching and Finite Automata Theory.
McGraw-Hill, New York, 1970.

[45] Koller, K.W. and U. Lauther.
The Siemens-AVESTA System for Computer Aided Design of MOS Standard-Cell Circuits.
In *Fourteenth Design Automation Conference.* I.E.E.E., June, 1977.

[46] Kowalski, T. J. and D. E. Thomas.
The VLSI Design Automation Assistant: Prototype System.
In *Proceedings of the 20'th Design Automation Conference.* IEEE, June, 1983.

[47] Kuhn, H. W.
The Hungarian Method for the Assignment Problem.
Naval Research Logistics Quarterly 2(1 & 2):83-97, March-June, 1955.

[48] Kuhn, L., S. E. Schuster, P. S. Zory, G. W. Lynch, and J. T. Parrish.
Experimental Study of Laser Formed Connections for LSI Wafer Production.
IEEE Journal of Solid-State Circuits SC-10(4):219-228, August, 1975.

[49] Kung, H.T.
Let's Design Algorithms for VLSI Systems.
In *Proceedings of the Caltech Conference on VLSI*, pages 65-90. California Institute of Technology, January, 1979.
Also available as a CMU Computer Science Department technical report, September 1979.

[50] Kung, H.T.
Why Systolic Architectures?
Computer Magazine 15(1):37-46, January, 1982.

[51] Kung, H.T. and Leiserson, C.E.
Systolic Arrays (for VLSI).
In Duff, I. S. and Stewart, G. W. (editors), *Sparse Matrix Proceedings 1978*, pages 256-282. SIAM, 1979.
A slightly different version appears in *Introduction to VLSI Systems* by C. A. Mead and L. A. Conway, Addison-Wesley, 1980, Section 8.3.

[52] Leiserson, C.E.
Area-Efficient VLSI Computation.
PhD thesis, Carnegie-Mellon University, 1981.

[53] Lesk, M. E. and E. Schmidt.
Lex — A Lexical Analyzer Generator.
In *UNIX Programmer's Manual 2*, chapter 20. Bell Telephone Laboratories, Inc., 1979.

[54] Lopez, A. D. and H.-F. S. Law.
A Dense Gate Matrix Layout Method for MOS VLSI.
IEEE Journal of Solid-State Circuits SC-15(4):736-740, August, 1980.

[55] B. Lowerre.
The Harpy Speech Recognition System.
PhD thesis, Carnegie-Mellon University, Computer Science Department, 1976.

[56] Lyon, R. F.
A Bit-Serial VLSI Architectural Methodology for Signal Processing.
In J. P. Gray (editor), *VLSI 81*, pages 131-140. University of Edinburgh, University of Edinburgh, August, 1981.

[57] Mead, C.A. and L.A. Conway.
Introduction to VLSI Systems.
Addison-Wesley, Reading, Massachusetts, 1980.

[58] Mead, C.A., Pashley, R.D., Britton, L. D., Daimon, Y.T. and Sando, S.F.
128-Bit Multicomparator.
IEEE Journal of Solid-State Circuits SC-11(5):692-695, October, 1976.

[59] Moore, E. F.
Gedanken Experiments on Sequential Machines.
In C. Shannon (editor), *Automata Studies*, pages 129-153. Princeton University Press, Princeton, N. J., 1956.

[60] Mukhopadhyay, A.
Hardware Algorithms for Nonnumeric Computation.
IEEE Transactions on Computers C-28(6):384-394, June, 1979.

[61] Muller, D.
Private Communication.

[62] North, J. C. and W. W. Weick.
Laser Coding of Bipolar Read-Only Memories.
IEEE Journal of Solid-State Circuits SC-11(4):500-505, August, 1976.

[63] Papert, S. and McNaughton, R.
Counter Free Automata.
MIT Press, Cambridge, 1973.

[64] Persky, G., D. N. Deutch, and D. G. Schweikert.
LTX — A Minicomputer-Based System for Automated LSI Layout.
Journal of Design Automation and Fault-Tolerant Computing 1(3):217-255, May, 1977.

[65] Philipson, L.
Private Communication.

[66] Rosenberg, A. L.
The Diogenes Approach to Testable Fault-Tolerant Arrays of Processors.
IEEE Transactions on Computers C-32(10):902-910, October, 1983.

[67] Salomaa, Arto.
Jewels of Formal Language Theory.
Computer Science Press, Rockville, Md., 1981.

[68] Savage, J. E.
The VLSI Compilation Techniques: PLA's; Weinberger Arrays; and SLAP, a New Silicon Layout Program.
Technical Report CS-82-24, Brown University, October, 1982.

[69] Saxe, J. B.
Dynamic-Programming Algorithms for Recognizing Small-Bandwidth Graphs in Polynomial Time.
Technical Report CS-80-102, Computer Science Department, Carnegie-Mellon University, January, 1980.

[70] Siskind, J. M., J. R. Southard, and K. W. Crouch.
Generating Custom High Performance VLSI Designs From Succinct Algorithmic Descriptions.
In *Proceedings of the 1982 Conference on Advanced Research in VLSI*, pages 28-40. M. I. T., January, 1982.

[71] Smith, R. T., J. D. Chlipala, J. F. M. Bindels, R. G. Nelson, F. H. Fischer, and T. F. Mantz.
Laser Programmable Redundancy and Yield Improvement in a 64K DRAM.
IEEE Journal of Solid-State Circuits SC-16(5):506-513, October, 1981.

[72] Southard, J. R.
MacPitts: An Approach to Silicon Compilation.
Computer 16(12):74-82, December, 1983.

[73] Steele, G. L.
Private Communication.

[74] Syphard, A. D. and N. D. Salman.
Automatic Laser Encoding of Semiconductor Read-Only Memories.
In Marlin O. Thurston (editor), *Proceedings of the National Electronics Conference, Volume XXIV*, pages 206-209. Illinois Institute of Technology, December, 1968.

[75] Trickey, H. W.
Good Layouts for Pattern Recognizers.
IEEE Transactions on Computers C-31(6):514-520, June, 1982.

[76] Ullman, J. D.
Combining State Machines and Regular Expressions for Automatic Synthesis of VLSI Circuits.
Technical Report CS-82-927, Stanford University, September, 1982.

[77] Valiant, L. G.
Universality Considerations in VLSI Circuits.
I.E.E.E. Transactions on Computers C-30(2):135-140, February, 1981.

[78] A. Weinberger.
Large Scale Integration of MOS Complex Logic: A Layout Method.
IEEE Journal of Solid State Circuits SC-2, February, 1967.

[79] Werner, J.
The Silicon Compiler: Panacea, Wishful Thinking, or Old Hat?
VLSI Design 3(5):46-55, September/October, 1982.

[80] Wolf, W., J. Newkirk, R. Mathews, and R. Dutton.
Dumbo, A Schematic-to-Layout Compiler.
In R. Bryant (editor), *Proceedings of the Third CalTech Conference on Very Large Scale Integration*, pages 379-393. CalTech, Pasadena, Ca., March, 1983.

[81] Yannakakis, M.
A Polynomial Algorithm for the Min Cut Linear Arrangement of Trees.
In *Proceedings of the 24'th Annual Symposium of Foundations of Computer Science*, pages 274-281. IEEE, November, 1983.